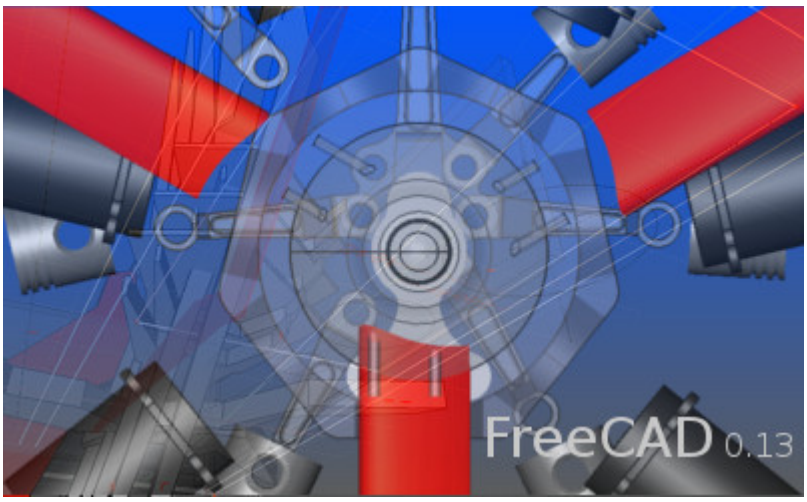
Manual02/fr

De FreeCAD Documentation

Manuel

de

FreeCAD



Ceci est le Manuel de FreeCAD. Il comprend les parties essentielles de la Page de garde de la documentation wiki.
Cette page est spécialement destinée à l'impression, comme un gros document, donc, si vous lisez ceci en ligne, vous pourrez préférer aller directement à la version **Aide en ligne**, qui est plus facile à parcourir.

Scripts et macros

Macros

Une **macro** est un moyen pratique et facile de créer une série de commandes dans FreeCad.

Il suffit d'enregistrer la série de commandes que vous faites, puis de sauver cet enregistrement sur disque en lui donnant un nom. Une fois cet enregistrement (macro) sauvé, vous pourrez l'exécuter autant de fois que vous le voulez.

Ces macros sont en réalité une liste de commandes écrites en langage python (http://fr.wikipedia.org/wiki/Python_(langage)), vous pouvez également les modifier, et créer des scripts très complexes.

Fonctionnement

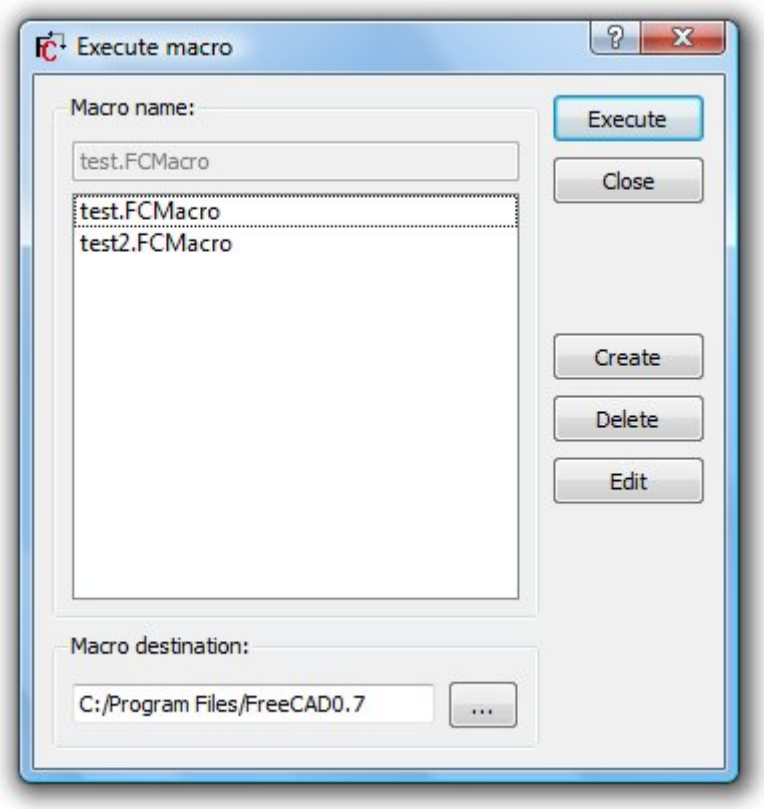
Si vous cochez dans menu **Édition → Préférences → Général → Macro → Montrer les commandes du script dans la console Python** , vous verrez dans la fenêtre " **Console Python** " que chaque action que vous exécutez s'affiche, par exemple en appuyant sur " **Afficher la vue de face** ", il s'affiche dans la console **Gui.activeDocument().activeView().viewFront()** qui est le code python correspondant.

Toutes ces commandes peuvent être enregistrées dans une macro.

Les commandes, qui servent à faire les macros, se trouvent sur la barre d'outils des macros :  .
Sur la barre d'outils, il y a 4 boutons: enregistrement, arrêt de l'enregistrement, édition de la macro, et, exécuter la macro.

Il est extrêmement facile d'utiliser ces commandes : dès que vous appuyez sur le bouton d'enregistrement, il vous est demandé de donner un nom à la macro, éventuellement, donnez l'emplacement où placer le fichier. Une fois que la macro est terminée, cliquez sur le bouton stop, et, toutes les actions que vous avez effectuées sont enregistrée. Pour exécuter la macro, cliquer sur le bouton d'édition et la boîte de dialogue **Lancer**

la macro s'affiche.



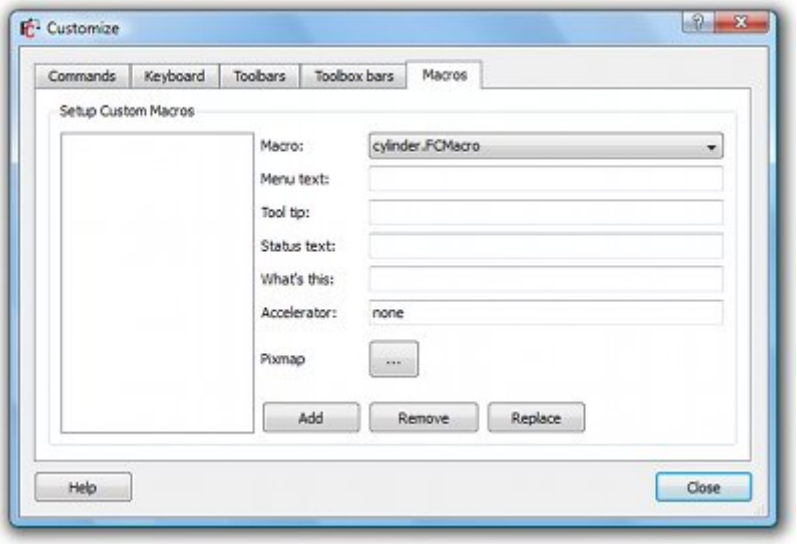
Vous pouvez ici gérer les macros enregistrées, lancer, créer, supprimer ou éditer une macro. L'édition ou la création d'une macro ouvre une nouvelle fenêtre dans FreeCad et vous pouvez ainsi créer ou modifier le code de la macro éditée.

Exemple

Cliquez sur le bouton d'| Enregistrement |, donnez un nom à la macro par exemple "cylinder 10x10" puis dans l'atelier Part, créez un cylindre de rayon = 10 et hauteur = 10. Puis cliquer sur le bouton | Stop | pour arrêter la macro. Dans la fenêtre d'édition de la macro vous pouvez voir le code en langage python qui a été enregistré et si vous le désirez, en modifier le code. Exécutez votre macro simplement en cliquant sur le bouton | Exécuter la macro dans l'éditeur |. La macro éditée ou la nouvelle macro est toujours sauvegardée lors de l'exécution, de manière à ne pas perdre les modifications apportées, les macros créées sont toujours accessibles à chaque nouvelle ouverture de FreeCad.

Personnalisation

Bien sûr, il n'est pas pratique de charger une macro dans l'éditeur en vue de l'exécuter. FreeCad fournit d'autres moyens pour exécuter votre macro, vous pouvez assigner un raccourci clavier à chaque macro ou créer un bouton de lancement sur la barre de menus. Une fois votre macro créée, ces raccourcis peuvent être crées par **Outils → personnaliser → Macros**



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

Création de macros sans enregistrement

Il est aussi possible d’insérer le code python d'une macro avec copier/coller sans enregistrement d'actions dans l'interface graphique. Créer simplement le code python de la macro, éditez-le, copiez-le et collez votre code directement dans l'éditeur de macros de FreeCad. Puis vous pouvez la réutiliser comme bon vous semble et la retrouver dans le répertoire réservé aux macros, en passant par **Macro → Macros** ou l'éditeur de macros sur la barre de menus  .

Référence sur les Macros

Visitez la page Recettes Macros pour charger des macros et les

ajouter à votre installation FreeCad. L'emplacement des macros est visible en cliquant sur l'icône de l'éditeur de macros  et, en bas de la boîte de dialogue **Destination de la macro**.

< précédent: Standard Menu suivant: Introduction to Python >
Index

Introduction à Python

Ceci est un petit tutoriel créé pour ceux qui veulent débuter en programmation Python (http://fr.wikipedia.org/wiki/Python_%28programming_language%29), qui est un langage de programmation (http://fr.wikipedia.org/wiki/Programming_language) open-source et multiplate-forme. Python a de nombreuses fonctionnalités qui le différencie des autre langages de programmation, et est facilement accessible à celui qui veut se lancer dans la programmation.

- Il a été conçu spécialement pour être facile à lire par les êtres humains, il est ainsi facile à apprendre et à comprendre
- Il est interprété. C'est-à-dire que contrairement aux langages compilés comme le C, votre programme n'a pas besoin d'être compilé pour être exécuté. Le code que vous écrivez peut être directement exécuté, une ligne après l'autre si vous le souhaitez. Cela permet de l'apprendre et de trouver les erreurs dans votre code facilement, parce que vous avancez lentement, une étape après l'autre.
- Il peut être intégré dans d'autres programmes comme langage de script. FreeCAD possède un interpréteur Python intégré, vous pouvez ainsi y écrire du code Python. Cela permet de manipuler des éléments de FreeCAD, par exemple de créer des objets géométriques. Il s'agit d'une fonction extrêmement puissante, parce qu'au lieu de se contenter de cliquer sur un bouton appelé "Créer une sphère", qu'un programmeur aurait placé là pour vous, vous avez la possibilité de créer simplement vos propres outils pour générer exactement les objets géométriques que vous souhaitez.
- Il est extensible; vous pouvez simplement installer de nouveaux modules Python et étendre ses fonctionnalités. Par exemple, il existe un module qui permet à Python de lire et d'écrire des images jpg, de communiquer avec twitter, de planifier des tâches exécutées pour votre système d'exploitation, etc...

Et maintenant au travail ! Soyez conscient que ce qui suit est une introduction simplifiée, et en aucun cas un tutoriel complet. Mais nous espérons qu'après cette lecture vous aurez acquis les bases nécessaires pour connaître et exploiter plus profondément les

mécanismes de FreeCad.

L'interpréteur

Habituellement, lors de l'écriture d'un programme informatique, il suffit d'ouvrir votre environnement de programmation préféré qui, est dans la plupart des cas, un éditeur de texte avec plusieurs outils autour de lui, écrire votre programme, puis le compiler et l'exécuter. Lorsque vous avez fait des erreurs pendant l'écriture, votre programme ne fonctionnera pas! et vous obtiendrez un message d'erreur vous indiquant ce qu'il s'est passé. Ensuite, vous revenez à votre éditeur de texte, corrigez les erreurs, exécutez à nouveau, et ainsi de suite jusqu'à ce que votre programme fonctionne parfaitement.

En Python, tout ce processus, peut être exécuté de manière transparente dans l'interpréteur Python. L'interpréteur Python est une fenêtre avec une invite de commande, vous pouvez simplement y taper votre code Python. Si vous installez Python sur votre ordinateur (téléchargez (http://www.python.org /download/) le depuis le site web Python si vous êtes sous Windows ou Mac, installez le à partir des gestionnaire de paquets, si vous êtes sous GNU / Linux), vous aurez l'interpréteur Python dans votre menu de démarrage. Mais FreeCAD dispose également d'un interpréteur Python intégré, vous n'êtes donc pas obligé de l'installer, cet interpréteur est visible dans la fenêtre inférieure (Si vous ne voyez pas cette fenêtre, cliquez sur **Affichage->Vues->Console Python**). Tous ces exemples ont été relu à partir de l'interpréteur disponible dans FreeCad.

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

L'interpréteur affiche la version de Python installée, puis le symbole **>>>**, qui est l'invite de commande pour entrer votre code Python. L'écriture du code dans l'interpréteur est très simple: une ligne, est une instruction. Lorsque vous appuyez sur ENTREE, votre ligne de code est exécuté (après avoir été instantanément compilé et cela de manière transparente pour vous).

Par exemple, écrivez ce code:

```
print "hello"
```

Ici **print** est une commande spéciale de Python qui signifie: affiche ce que je te demande. Lorsque vous pressez ENTREE, l'opération s'exécute et le message "bonjour" s'affiche à l'écran. Si vous effectuez une erreur, par exemple, écrivez:

```
print hello
```

Python vous dira qu'il ne sait pas ce qu'est bonjour. Les caractères **"** (guillemets) spécifient que le contenu est une chaîne de caractères qui doit être affichée. Sans les **"** (guillemets), la commande d'affichage de bonjour n'est pas reconnue comme du texte, mais comme un mot-réservé spécial de Python. L'important est, que vous obtenez immédiatement une notification d'erreur. En appuyant sur la flèche Haut (ou, dans l'interpréteur FreeCAD, CTRL + flèche Haut), vous pouvez revenir à la dernière commande que vous avez écrite et la corriger.

L'interpréteur Python dispose également d'un système d'aide intégré. Voulez vous taper:

```
help
```

ou, par exemple, nous n'avons pas compris ce qui n'allait pas avec notre commande d'affichage "help" ci-dessus, et nous allons demander des informations spécifiques sur la commande "print":

tapez

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

Nous voilà devant une longue description sur la commande "print".

Maintenant, nous dominons totalement notre interpréteur, et nous pouvons commencer à travailler sérieusement.

Les Variables

Bien sûr, vous vous dites que l'affichage de "bonjour" n'est pas très intéressant. Il peut y avoir alors des choses plus intéressantes comme par exemple, l'affichage de choses que vous ne savez pas, et laisser Python trouver ces choses pour vous. C'est là que le concept de **"variable"** entre en jeu. Une variable est tout simplement une valeur que vous stockez en mémoire avec un nom identificateur. Par exemple, tapez ceci:

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

Avez vous compris ce qui s'est passé ? Nous avons «sauvé» en mémoire la chaîne "bonjour" dans la variable qui porte le nom de **a**. Maintenant, **a** n'est plus un nom inconnu ! Nous pouvons maintenant l'utiliser n'importe où, comme par exemple dans la commande d'affichage à l'écran **print**. Nous pouvons dans Python utiliser n'importe quel nom que nous voulons, tout en respectant de simples règles, comme, ne pas utiliser d'espaces ou de signes de ponctuation. Par exemple, nous pouvons écrire:

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

Compris ? maintenant **hello** n'est plus un mot inconnu. Que faire alors si, par inattention ou par méprise nous choisissons un nom qui existe dans Python? Admettons que nous voulons stocker notre chaîne sous le nom de "print":

```
print = "hello"
```

Python se rend compte immédiatement de l'erreur et vous signale qu'il est impossible de donner ce nom à votre variable. Il y a quelques restrictions dans Python, les mots "réservés" ne peuvent pas être modifiés! Mais, nos propres variables peuvent être modifiées à tout moment, c'est exactement pour cela qu'elles sont appelées **variables**, le contenu de la variable peut varier. Par exemple:

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

Nous venons de changer la valeur de **myVariable**. Nous pouvons également copier des variables:

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

Notez qu'il est judicieux de donner des noms descriptifs à vos variables, lorsque vous écrivez un long programme, vous ne saurez plus à quoi sert votre variable "**a**". Mais, si vous la nommez, par exemple **MonMessageDeBienvenue**, vous vous souviendrez facilement a quoi vous l'aviez destinée quand vous la verrez.

Plus de renseignements sur les variables Python (http://fr.wikibooks.org/wiki/Programmation_Python/Variable)

Les Nombres

Vous savez qu'un programme informatique est utilisé pour traiter toutes sortes de données, non seulement du texte mais aussi et surtout des nombres. Une des choses les plus importantes dans Python, est que Python doit savoir quel type de données seront traitées. Nous avons vu dans notre exemple d'affichage "bonjour" que la commande d'affichage **print** a reconnu «bonjour» comme

une chaîne. C'est grâce au " **"** " (guillemets), que la commande d'affichage **print** sait qu'il va traiter une chaîne de caractères alphabétiques (du texte).

Le type de donnée contenu dans une variable peut être connu à n'importe quel moment grâce à la commande spéciale de Python **type()**:

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

Dans cet exemple, il s'affiche dans la console Python **<type 'str'>** dans le langage informatique on dit qu'il est de type "string" (chaîne de caractères alphabétiques). Il y a d'autres types (http://fr.wikibooks.org/wiki/Programmation_Python/Type) de données, par exemple: les nombres entier **(integer)** , les nombres à virgule flottante **(float)** . . .:

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

C'est déjà plus intéressant, n'est-ce pas? Maintenant nous avons une puissante calculatrice! Voyons maintenant comment elle fonctionne. Python sait que 10 et 20 sont des nombres entiers. Donc, ils sont stockés en mémoire sous forme **"int"** (integer), et Python peut travailler avec eux comme il peut le faire avec des nombres entiers. Regardez les résultats de ce code:

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

Vu ? Nous avons forcé Python à considérer nos deux variables non pas comme de simples nombres, mais comme des parties de texte. Python peut concaténer deux parties de texte, mais il ne cherchera pas à trouver leur somme. Nous avons parlé de nombres entiers, il y a aussi des nombres à virgule flottante. La différence est, que les nombres entiers n'ont pas de partie décimale, alors que les nombres à virgule flottante peuvent avoir une partie décimale:

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

Les types **entier** et à **virgule flottante**, **Int** et **Float** peuvent être mélangés sans problème:

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

Naturellement, la somme comporte des décimales, vrai? Pendant l'opération, Python automatiquement a décidé que le résultat serait un type **Float** (virgule flottante). Dans certains cas comme celui-ci, Python détermine automatiquement quel type doit être choisi pour un résultat. Dans d'autres cas il déclanchera une erreur. Par exemple:

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

Dans cet exemple , varA est une **chaîne** et varB est un **int**, Python ne mélange pas les types différents et nous donnera une erreur. Mais, nous pouvons forcer Python a mélanger des types différents grâce à la conversion:

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

Maintenant, l'opération fonctionne, pourquoi ! Vous avez noté, que nous avons **converti** varB en "string" ***au moment de l'affichage*** avec la commande **str()**, mais nous n'avons pas modifié le type de varB qui reste un **int**. Si nous voulons convertir varB de façon permanente en une chaîne de caractères pour les besoins futur du programme, nous aurons besoin de faire:

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

Nous pouvons également utiliser les commandes **int()** et **float()**

pour convertir une chaîne de caractères **str** en un **int** ou **float** Pour la conversion, il faut faire:

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

Note au sujet des commandes Python

Vous avez sûrement remarqué que dans cette partie du tutoriel, nous avons utilisé la commande d'affichage **print** de plusieurs manières. Nous avons affiché des variables, des opérations, des chaînes séparées par des virgules et même le résultat de la commande Python **type()**. Peut-être avez vous également remarqué qu'en faisant ces deux commandes,

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

nous obtenons le même résultat.

Tout s'affiche automatiquement à l'écran parce que nous sommes dans l'interpréteur. Lorsque nous allons écrire des programmes plus complexes qui s'exécuteront hors de l'interpréteur, ils ne seront pas affichés à l'écran, pour les afficher nous aurons besoin d'utiliser la commande **print**. Mais maintenant, nous allons cesser de l'utiliser pour augmenter la vitesse d'exécution.

Donc, nous allons simplement écrire:

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

Attention, Python est sensible à la casse, ***myVar*** est différent de ***myvar*** !!!
Vous avez remarqué que la plupart des commandes Python (ou mots-réservés) que nous connaissons ont des parenthèses, qui sont utilisées pour dire avec quoi la commande doit travailler: type(), int(), str(). . . etc. La seule exception est la commande **print**, qui en vérité ne l'est pas car, elle peut fonctionner aussi bien avec ou sans parenthèses.

Exemple:

```
print ("bonjour")
print "bonjour"
```

Les Listes (Tableaux)

Un autre type de données intéressant, est le type **list**. Le type **list** est simplement une liste de données. De la même manière que nous définissons une chaîne de texte en utilisant **" "** (guillemets), nous définirons des listes en utilisant **[ ]** (crochets):

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

Vous voyez qu'une liste peut contenir n'importe quel type de données. Les listes sont très utiles car vous pouvez grouper des variables ou des données ensembles. Vous pouvez alors faire toutes sortes de choses au sein de ces groupes, par exemple, les compter avec **len()**:

```
len(myOtherList)
```

ou récupérer un objet de cette liste:

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

Vous voyez que la commande **len()** renvoie le nombre d'éléments dans une liste, la «position» d'un objet dans la liste commence à 0. Le premier élément dans une liste est toujours à la position 0, donc dans notre myOtherList, "Bob" est a la deuxième position. Nous pouvons faire beaucoup plus de choses avec les listes tel que le tri du contenu, la suppression ou l'ajout d'éléments d'autres renseignements sur List (http://www.diveintopython.net/native_data_types/lists.html).

Une chaîne de texte est très semblable à une liste et chaque

caractère peut être adressé séparément! Essayez ce code:

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

Pratiquement, ce que vous faites avec les listes peut également être fait avec les chaînes de caractères. En fait, les listes et les chaînes de caractères sont des séquences que Python voit en interne de la même manière.

Outre les chaînes de caractères **"String"**, les entiers **"Integer"**, les nombres à virgule flottante **"float"** et les listes **"list"**, il y a beaucoup de type de données, plus de renseignements sur les dictionnaires (http://www.diveintopython.net/native_data_types/index.html#odbchelper.dict). Vous pouvez même créer vos propres types de données avec des classes (http://www.freenetpages.co.uk/hp/alan.gauld/tutclass.htm).

L'Indentation

Une manière pratique et élégante d'afficher chaque élément de la liste, est de naviguer à l'intérieur de cette liste.
Entrez ce code dans la console:

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

Nous venons de faire une "itération" (encore un nouveau mot de programmeur!) grâce à notre boucle " **for ... in ... :** " nous avons scruté chaque "champ" de la variable **alldaltons**. Notez la syntaxe particulière de la boucle, la commande se termine avec un " **:** " ce qui indique à Python que la suite sera un bloc d'une ou plusieurs commandes ou instructions.

Après avoir frappé ENTREE derrière le " **:** ", l'invite de commande va changer en " **...** " ce qui indique à Python que la suite sera une partie de celui-ci.

Alors comment savoir, combien de ligne(s) sera ou seront

exécutées par Python à l'intérieur de la boucle ? Pour créer un bloc, Python utilise l'indentation. Les prochaines lignes ne commenceront pas au prompt " **>>>** " mais elles commenceront par un ou plusieurs espaces vides, ou, une ou plusieurs tabulations. Les langages de programmation utilisent leurs propres méthodes , comme, la mise entre parenthèses du bloc, entre un BEGIN ... END etc.

Tant que vous écrirez vos lignes avec la même indentation, elles seront considérées comme faisant partie du bloc. Si vous commencez une ligne avec 2 espaces et la prochaine avec 4 espaces, il y aura une erreur. Lorsque vous avez terminé votre bloc, il suffit d'écrire la suite du programme sans indentation, ou appuyez simplement sur Entrée.

Créer des indentations permet aussi d'éclaircir la lecture code dans le cas de grands programmes. Nous allons voir que de nombreuses autres commandes indentées peuvent avoir des blocs de code aussi.

- **>>>** alldaltons = ["Joe", "William", "Jack", "Averell"] ENTREE
- **>>>** for dalton in alldaltons**:** ENTREE
- **...** ESPACE ESPACE print dalton + " Dalton" ENTREE
- **...** ENTREE
- **>>>**

La commande " **for ... in ... :** " peut être utilisée pour de nombreuses procédures qui doivent être effectuées plus d'une fois (en boucle). Elle peut aussi par exemple être combinée avec la commande **range()**:

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

Ou des choses plus complexes comme ceci:

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

Vous voyez que la commande **range()** a également la particularité de commencer à 0 (si vous ne spécifiez pas un nombre de départ) et que son dernier nombre sera le nombre que vous aurez spécifié **moins un** . Bien sûr, cette commande fonctionne parfaitement avec les autres commandes Python.

Par exemple:

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

Une autre fonction intéressante utilisée dans un bloc indenté est la commande de condition **if** (si). Avec " **if** " la suite de la procédure sera exécutée **uniquement si la condition est remplie**.

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

C'est bien, ce code affiche "OK il c'est bien un Dalton !!!" car la condition est exacte. Mais maintenant essayons cette ligne:

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

Il ne c'est rien affiché car la condition n'était pas remplie. Nous pouvons alors lui demander **else** (si la condition n'est pas remplie alors):

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

Les Fonctions

Il n'y a pas beaucoup mots réservés dans Python (http://docs.python.org/reference/lexical_analysis.html#identifiers), à peine une trentaine, et nous en connaissons maintenant quelques unes. Imaginons que nous voulions construire nous même une commande spéciale! Et bien, il est extrêmement facile de construire sa propre commande dans Python. Vous pouvez ajouter ces commandes dans votre installation Python de manière à en augmenter les capacités et les utiliser comme bon vous semble. Ces nouvelles commandes que vous allez créer dans Python, s'appellent des **Fonctions**. Elles sont faites de cette manière:

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

printsqm(45)
```

Extrêmement simple ! Le mot réservé "**def()**" crée une nouvelle fonction dans Python. Vous lui donnez un nom, dans l'exemple: "**printsqm**". Dans les parenthèses, la variable qui va transmettre les données à la fonction, dans l'exemple: "**myValue**". A l'intérieur de la fonction (donc après le " **:** " et une indentation) , vous définissez les formules, les données ou tout ce que vous voulez transformer et que la fonction va vous retourner.

Par exemple, regardez la commande (ou mot réservé) **len()**. Si vous écrivez len() simplement, Python affichera "**TypeError: len() takes exactly one argument (0 given)**" il vous dit, vous voulez len() de quelque chose alors j'ai besoin d'un argument pour l'exécuter ! Puis, par exemple, vous allez écrire len("William") et vous en obtiendrez la longueur. Alors, "William" ou une variable est un argument que vous passez à la len(). La fonction len() est définie de telle manière qu'elle sait exactement quoi faire avec l'argument qui lui a été transmis.

Le nom de la variable "**myValue**" peut être n'importe quel nom, et cette variable ne sera utilisée qu'à l'intérieur de la fonction. C'est juste un nom qui représentera l'argument dans la fonction en vue de l'utiliser, mais elle sert aussi a renseigner la fonction de combien d'arguments elle disposera. Par exemple, faites ceci:

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

Cette commande affichera l'erreur "**TypeError: printsqm() takes exactly 1 argument (2 given)**" car la fonction "**def printsqm(myValue):**" ne demande qu'un seul argument, "**myValue**" et, nous lui en avons donné deux, 45 et 34.

Maintenant, écrivez cette fonction:

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

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

Nous avons créé une fonction qui demande deux arguments, les exécutes , et nous renvoie le résultat. Le retour du résultat est très utile car nous pouvons l'utiliser pour l'afficher ou le stocker dans une variable **myTotal** (pour notre exemple mais n'importe quel nom conviendra) ou les deux. Comme nous sommes dans l'interpréteur de Python, le résultat s'affiche en faisant:

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

Mais une fois le programme terminé et exécuté hors de l'interpréteur il n'y aura pas d'affichage ! Pour afficher le résultat hors de l'interpréteur Python, il faut bien sûr utiliser la commande **print**. Alors il faudra faire:

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

Voilà c'est affiché.

Pour plus de renseignements sur les autres possibilités des fonctions (http://www.diveintopython.net/getting_to_know_python/declaring_functions.html).

Les Modules

Maintenant, vous avez une idée du fonctionnement de Python: mais comment faire pour travailler avec les fichiers et les modules.

Jusqu'à présent, nous avons écrit des instructions ligne par ligne pour travailler dans l'interpréteur Python, pas vrai? Lorsque vous voulez faire des choses plus complexes, il est commode d'écrire les premières lignes de code, puis de les exécuter en une seule fois. Eh bien, c'est très facile à faire, et cela permet aussi de sauver son travail. Il suffit d'ouvrir un éditeur de texte (par exemple, Le Bloc-notes Windows), et d'écrire toutes les lignes de code de Python, de la même manière qu'elles sont écrites dans l'interpréteur, avec les indentations, etc. Ensuite, enregistrez le fichier sur votre disque, de préférence avec l'extension **.Py**.

Voilà, maintenant vous avez un programme Python complet. Bien sûr, il y a de meilleurs éditeurs que le bloc-notes de Windows ou le terminal (http://www.osxfacile.com/terminal.html) d'OS X comme l'excellent Notepad++ (http://notepad-plus-plus.org/fr/) (pour Windows) qui utilise la coloration syntaxique tout comme XCode (https://developer.apple.com/xcode) (pour OS X) et ceci démontre qu'un programme Python n'est qu'un fichier texte.

Pour exécuter un programme Python, il ya des centaines de manières. Dans Windows, cliquez simplement sur le fichier, ouvrez-le avec Python, et exécutez le. Mais vous pouvez également l'exécuter avec l'interpréteur Python. Pour ce faire, l’interpréteur doit savoir où se trouve le programme **.Py**. Dans FreeCAD, le plus simple est de placer les fichiers **.Py** dans le répertoire par défaut destiné aux programmes Python, cet endroit connu de l'interpréteur inclut dans FreeCAD est ***C:\Program Files\FreeCAD0.12*\bin**, mais d'autres endroits sont aussi connu de FreeCad ***C:\Program Files\FreeCAD0.12*\Mod** (tous les outils de FreeCad) et ***C:\Travail\Mes documents*\. . .\FREECAD\Macro** où sont répertoriés tous vos programmes créés dans l’interpréteur de FreeCad **Macro-->Macros**. Le chemin de destination de vos modules peut être forcé à partir du menu ***Édition-->Préférences-->Macro*** **Chemin de la macro**.

Supposons que nous écrivions ce fichier programme:

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

print "test.py succesfully loaded"
```

et, nous allons l'enregistrer en "**test.py**" dans **. . ./FreeCAD/bin**.

Maintenant, allons dans FreeCAD, et dans la fenêtre de l'interpréteur, écrivez:

```
import test
```

sans l'extension .py.

Le contenu du fichier sera tout simplement exécuté, ligne par ligne, comme si nous l'avions écrit dans l'interpréteur. La fonction somme a été créée, et le message "***test.py a bien été chargé***" sera affiché. Il ya une grande différence: la commande **import** est faite non seulement pour exécuter des programmes écrits dans des fichiers comme le nôtre, mais aussi de charger des fonctions dans Python, de sorte qu'elles deviennent disponibles dans l'interpréteur. Les fichiers contenant des fonctions, comme le nôtre, sont appelés **modules**.

Normalement, lorsque nous écrivons une fonction sum() dans l'interpréteur, nous l'exécutons simplement comme ceci,

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

comme nous l'avons fait plus haut.

Mais quand nous importons un module contenant une fonction comme **sum(a,b)**, la syntaxe est un peu différente. Nous ferons:

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

Autrement dit, le module est importé comme un «conteneur», et toutes ses fonctions sont à l'intérieur. Cela est extrêmement utile, parce que nous pouvons importer un grand nombre de modules,

et de les organiser.
Donc, en bref, quand vous voyez **quelque_chose.quelque_chose** (avec un **point** entre les deux), signifie que quelque chose est à l'intérieur quelque chose.

Nous pouvons aussi, importer et extraire notre fonction sum() contenue dans "test.py" directement dans l'interpréteur, comme ceci:

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

Théoriquement, tous les modules se comportent de cette manière. Vous importez un module, et vous utilisez ses fonctions de cette manière: **module.fonction(argument(s))**.
Les modules travaillent de cette façon: ils définissent les fonctions, les nouveaux types de données et les classes que vous pouvez utiliser dans l'interpréteur Python ou dans vos propres modules, parce que rien ne vous empêche d'importer des modules à l'intérieur de votre module!

Encore une chose extrêmement utile. Comment connaître les modules disponibles ? quelles sont les fonctions contenues dans ces modules et comment les utiliser (c'est à dire quels arguments sont demandés par la fonction)? Nous avons vu que Python a une fonction d'aide().

Alors, dans l'interpréteur Python de FreeCad faisons:

```
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)
```

Nous voyons maintenant toutes les fonctions contenues dans le module math, ainsi que des trucs étranges comme: __ *doc*__, __

FILE__, __ name__ . . . .
Le **__ doc__** est extrêmement utile, il s'agit d'un texte de documentation. Dans les modules, chaque fonction de fait a une **__ doc__** qui explique comment l'utiliser. Par exemple, nous voyons qu'il ya une fonction **sin** dans le module math.
Vous voulez savoir comment utiliser cette fonction ? alors:

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

Et enfin, une dernier chose: Lorsque l'on travaille sur un nouveau module, nous avons besoin de le tester. Donc, une fois que nous avons écrit une partie du code, dans l'interpréteur Python, nous ferons:

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

Mais que faire, si myTestFunction() ne fonctionne pas correctement? Nous retournons à notre éditeur et nous le corrigeons. Puis, au lieu de fermer et de rouvrir l'interpréteur python, nous allons tout simplement mettre à jour le module comme ceci:

```
reload(myModule)
```

Démarrer avec FreeCAD

Eh bien, je pense que maintenant vous devez avoir une bonne idée de la façon dont Python travaille, et vous pouvez commencer à explorer ce que FreeCAD peut nous offrir. Les fonctions Python de FreeCAD sont toutes bien organisées en différents modules. Certaines d'entre elles sont déjà chargées (importées) au démarrage de FreeCAD. Donc, il suffit de faire:

```
dir()
```

et lire dans l’interpréteur tous les modules chargés dans FreeCad, voir Scripts de base dans FreeCad...

Bien sûr, nous n'avons vu qu'une très petite partie de l'univers Python. Il existe de nombreux concepts importants que nous n'avons pas mentionné ici.
Voici deux liens de référence de Python sur le net:

- Le site officiel de Python (http://docs.python.org/reference/) (en)
- Plongez dans le Wikibook/ Book de Python (http://www.diveintopython.net) (en)
- Wiki en français (http://fr.wikibooks.org /wiki/Programmation_Python)
- Un autre aussi en français (http://www.jchr.be/python /index.htm)

Pensez à en faire des onglets !

< précédent: Macros Index suivant: Python scripting tutorial >

Python scripting in FreeCAD

FreeCAD a été programmé dès la première ligne de code dans le but d'être totalement contrôlé par des scripts écrits en Python. Presque toutes les procédures de FreeCAD, telles que l'interface, le contenu des scènes, même la représentation du contenu des vues 3D, sont accessibles à partir de l'interpréteur Python ou de vos propres scripts.
Par conséquence, FreeCAD est probablement l'une des applications d'ingénierie la plus profondément personnalisable et évolutive disponible actuellement.

Dans son état actuel, FreeCAD a très peu de commandes de base pour interagir avec vos objets 3D, FreeCAD est encore jeune et est encore au stade de développement, de plus, la philosophie du développement de FreeCAD est orientée de manière à fournir une plate-forme CAD plutôt qu'une application d'utilisation spécifique.
Grâce aux scripts Python utilisables dans FreeCAD, nous avons un moyen très simple et rapide de voir et de tester les nouvelles fonctionnalités des modules élaborés par la communauté internationale des utilisateurs, des utilisateurs qui, généralement connaissent la programmation Python.
Python est l'un des langages interprétés les plus populaires et, généralement considéré comme très facile à apprendre, bientôt, vous pourrez aussi écrire vos scripts pour modeler "votre propre" FreeCAD.

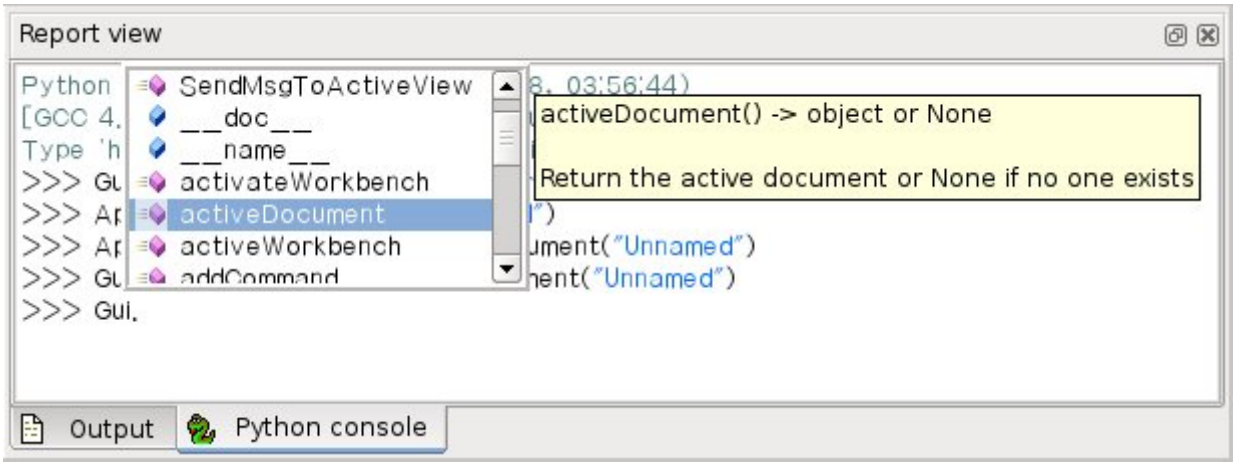
Si vous n'êtes pas familier avec Python, nous vous recommandons de chercher des tutoriels sur internet et "jeter un œil rapide" (http://python.50webs.com/) sur sa structure. Python est un langage très facile à apprendre, en particulier parce qu'il peut être exécuté à l'intérieur de l'interpréteur, de la plus simple commande jusqu'à l'élaboration de programmes complexes, il peut être exécuté à la volée sans avoir besoin de compilateur.
FreeCAD dispose de son propre interpréteur Python intégré. Si vous ne voyez pas de fenêtre intitulée **Console Python** comme illustré ci-dessous, vous pouvez l'activer en cliquant dans la barre d'outils **Affichage -> Vues -> Console Python** pour afficher

l’interpréteur Python.

L'interpréteur Python

A partir de l’interpréteur Python, vous pouvez accéder à l'ensemble des modules Python installés, les modules originaux de FreeCAD, ainsi que tous les modules supplémentaires que vous installerez plus tard dans FreeCAD. La capture d'écran ci-dessous vous montre l'interpréteur Python:

A partir de l’interpréteur, vous pouvez exécuter du code Python et naviguer à travers les classes et fonctions disponibles. FreeCAD fournit un navigateur de classe très pratique pour l'exploration de votre nouvel univers qu'est FreeCAD. Lorsque vous tapez le nom d'une classe connue suivie d'un "**.**" (point) (ce qui veut dire que vous voulez ajouter quelque chose après le point à partir de cette classe), une fenêtre s'ouvre et vous renseigne sur les options et méthodes disponibles dans cette classe. Lorsque vous sélectionnez une option, le texte d'aide qui lui est associé (s'il est disponible) est automatiquement affiché:



Alors, commencez ici en tapant **App.** ou **Gui.** (Attention à la casse **App** est différent de **app**) et regardez ce qui se passe.
Une autre façon plus simple d'explorer Python le contenu des modules et des classes est d'utiliser la commande d'affichage **dir()**.
Par exemple, en tapant **dir()** tous les modules actuellements répertoriés et chargés dans FreeCAD s'affichent.Si vous tapez **dir(App**) tout ce qu'il y a à l'intérieur du module **App** sera affiché , etc.

Une autre caractéristique utile de l'interprèteur est la possibilité de revenir en arrière dans l'historique des commandes et récupérer une ligne de code que vous avez tapé plus tôt. Pour naviguer dans l'historique des commandes, il suffit d'utiliser CTRL + HAUT ou CTRL + BAS.

Si vous cliquez avec le bouton droit de la souris dans la fenêtre de l'interpréteur, vous avez également les options classiques d'un traitement de texte, telles que copier tout l'histoire (utile lorsque vous voulez expérimenter votre code avant de faire votre script final), ou d'insérer un nom de fichier avec le chemin complet.

Aide Python

Dans le menu **Aide** de FreeCAD, vous trouverez une entrée portant la mention **Modules Python**, qui va ouvrir dans le navigateur une fenêtre contenant la liste complète, de la documentation de l'ensemble des modules Python à disposition de l’interpréteur FreeCAD, c'est à dire les modules fournis avec

Python et ceux intégrés dans FreeCAD. La documentation disponible dépend de l'effort que le développeur a mis pour documenter le code son module, les modules Python en général, ont la réputation d'être bien documentés. FreeCAD doit rester ouvert pour travailler avec ce système de documentation.

Modules incorporés (Built-in)

FreeCAD étant conçu pour être exécuté sans interface graphique (GUI), la quasi-totalité de ses fonctionnalités est séparé en deux groupes: les fonctionnalités de base, nommés **«App»**, et la fonctionnalité graphique, nommée **«Gui»**. Donc, nos deux principaux modules dans FreeCAD sont appelés **App** et **Gui**.
Ces deux modules peuvent également être accessibles à partir des scripts, respectivement avec les noms **FreeCAD** et **FreeCADGui**. Ils sont accessibles même hors de l'interpréteur.

- Dans l'**App module**, vous trouverez tout ce qui concerne l'application elle-même, comme, les procédures d'ouvrir ou fermeture de fichiers, comme l'ouverture de la feuille active ou lister le contenu de la feuille . . .

- Dans l'**Gui module**, vous trouverez des outils pour accéder et gérer les éléments graphiques, comme les boutons utilisateurs et leur barres d'outils, et, plus intéressant, la représentation graphique de l'ensemble du contenu FreeCAD.

Lister tout le contenu de ces modules est un contre-productif, car ils grandissent très vite compte tenu de la progression du développement de FreeCAD.
Mais les deux outils fourni (le navigateur de classe et de l'aide de Python) vous donnerons, à tout moment, une complète documentation mise à jour sur ces modules.

Les objets "App" et "Gui"

Comme nous l'avons dit, dans FreeCAD, tout est séparé entre le noyau et la représentation du projet. Y compris les objets 3D. Vous pouvez accéder aux propriétés des objets (appelés

fonctions dans FreeCAD) via le module **App**, et modifier la façon dont ils sont représentés sur l'écran via le module de **Gui**.
Par exemple, un cube possède des propriétés qui le définissent, (comme la largeur, longueur, hauteur) qui sont stockées dans un **App objet** et, les propriétés de représentation (comme la couleur des faces, le mode de dessin) qui sont stockées dans un objet correspondant **Gui**.

Cette méthode de travail permet une multitude d'utilisations, comme des algorithmes travaillant uniquement sur la partie caractéristiques, sans avoir à se soucier de la partie visuelle, voire de réorienter le contenu du document à une partie non-graphique de l'application, tels que des listes, des tableurs, ou l'analyse d'éléments.

Pour chaque objet App dans votre document, **il existe un objet correspondant Gui**.
En fait le document lui-même possède à la fois des **objets App** et des **objets Gui**. Bien sûr, ceci n'est valable que lorsque vous exécutez FreeCAD dans son interface graphique. Dans la version en ligne de commande (sans interface graphique), seuls les "objets App" sont accessibles.
Notez que la partie "objet Gui" est réactualisé chaque fois qu'un "objet App" est recalculé (par exemple lorsqu'il y a un changement de paramètres), les changements que vous pourriez avoir fait directement à l'objet Gui peuvent être perdues.

Pour accéder à la partie **App** d'un objet, vous devez tapez:

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

où "ObjectName" est le nom de votre objet.
Le même résultat est obtenu en tapant:

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

Pour accéder à la partie **Gui** d'un l'objet , vous tapez:

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

où "ObjectName" est le nom de votre objet.
Le même résultat est obtenu en tapant:

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

Si vous n'êtes pas dans l'interface graphique (Gui) (par exemple si vous êtes en mode ligne de commande), la dernière ligne retournée sera 'None'.

Les objets dans un document

Dans FreeCAD tout votre travail est dans un "Document". Ce document contient vos formes géométriquee et peut être sauvegardé dans un fichier. Dans FreeCAD, plusieurs documents peuvent être ouverts en même temps. Le document, et les formes géométriques contenues , sont des **objets App** et des **objets Gui**. Les objets App contiennent les définitions des formes géométriques réelles, tandis que les objets Gui contiennent les différentes vues de votre document.
Vous pouvez ouvrir plusieurs fenêtres, chacune de ces fenêtres peut afficher votre projet avec un facteur de zoom différent ou des vues différentes du projet. Ces vues font toutes partie de l'objet Gui de votre document.

Pour accéder à la partie App du document ouvert (actif), tapez:

```
myDocument = App.ActiveDocument
```

Pour créer un nouveau document, tapez:

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

Pour accéder à la partie graphique (Gui) du document ouvert (actif), tapez:

```
myGuiDocument = Gui.ActiveDocument
```

Pour accéder à la vue courante, tapez:

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

Modules supplémentaires

Les modules **FreeCAD** et **FreeCADGui** sont utilisés uniquement pour créer et gérer des objets dans le document FreeCAD. Ils ne sont pas utilisés pour la création ou la modification des formes géométriques.
Les formes géométriques peuvent être de plusieurs types, elles sont donc construites par des modules supplémentaires, chaque module s'occupe la gestion d'un type de forme géométrique spécifique.
Par exemple, le module "Part utilisé par le noyau OpenCascade, et donc capable de créer et manipuler des formes géométriques de type B-rep (http://fr.wikipedia.org /wiki/Boundary_representation), pour lequel OpenCascade est construit.
Le module "Mesh" est capable de construire et modifier des objets Mesh (mailles). De cette façon, FreeCAD est capable de gérer une grande variété de types d'objets, qui peuvent coexister dans le même document, et de nouveaux types d'objets pourront êtres ajoutés facilement et constamment.

Création d'objets

Chaque module a sa propre manière de gérer sa forme géométrique, mais il y a une chose qu'ils peuvent tous faire, c'est de créer des objets dans le document.
Mais, le document FreeCAD connaît tous les types d'objets disponibles fournis par les modules,
tapez:

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

FreeCAD listera tous les objets possibles que vous pouvez créer.
Par exemple, nous allons créer un objet maillage (traité par le module "Mesh") et une objet Part (traité par le module le "Part"):

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

Le premier argument est le type d'objet **"Mesh::"**, le second est le nom de l'objet **"myMeshName"**. Nos deux objets semblent identiques: Ils ne contiennent pas encore de forme géométrique, et la plupart de leurs propriétés sont les mêmes lorsque vous les inspecter avec **dir(myMesh)** et **dir(myPart)**.
Sauf que, myMesh a une propriété "Mesh" (maille) et myPart a une propriété "Part" (forme géométrique).
C'est de cette manière que les données de "Mesh" (maillage) et "Part" (forme géométrique) sont stockées.
Par exemple, nous allons créer un cube (Part) et le stocker dans notre objet myPart:

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

Si vous essayez de stocker le cube avec la propriété **objet Mesh "myMesh"**, il retournera une **erreur de type**. Car ces propriétés sont conçues uniquement pour stocker un type d'objet bien défini.
Dans la propriété **objet Mesh "myMesh"**, vous ne pouvez enregistrer que des objets créé avec le module **Mesh**.
Notez que la plupart des modules disposent également d'un raccourci pour ajouter leur formes géométriques au document:

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

Modification d'objets

La modification d'un objet est faite de la même manière:

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

Maintenant, nous allons construire un cube plus gros:

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

Questionner les objets

Vous pouvez toujours connaître de quel type est un objet.
Faites ceci:

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

ou de savoir si un objet fait partie d'un modèle de base (Part Feature, Mesh Feature, etc):

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

Retourne **TRUE** ou **FALSE**

Maintenant vous pouvez commencer à travailler avec FreeCAD! Pour savoir ce que vous pouvez faire avec le **Part Module**, lisez la page Part scripting, ou la page Script Mesh pour travailler avec le module Mesh .
Notez que, bien que les modules **Part** et **Mesh** sont les plus complets et les plus largement utilisés, les autres modules tels que le Draft Module (Projet) ont également leurs API scripts qui peuvent vous être utiles.
Pour une liste complète de chaque module et de leurs outils disponibles, visitez la section :Category:API (en).

< précédent: Python scripting tutorial suivant: Mesh Scripting >
Index

Introduction

Avant de commencer, vous devez importer le module **Mesh**. Tapez (*Attention à la classe **Mesh** est différent de **mesh***):

```
import Mesh
```

Dès que vous avez importé le module de maillage de la classe Mesh, vous accéderez facilitent aux fonctions C++ Mesh-Kernel de FreeCAD.

Création et chargement

Pour créer un objet maillage vide il suffit d'utiliser la commande standard:

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

Vous pouvez aussi créer un objet à partir d'un fichier

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

Une liste de fichiers compatibles avec "Mesh" (maillage) est disponible **ici**.

Ou de créer un ensemble de triangles en les décrivants par leurs sommets (Vertex):

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

Le kernel-Mesh prend soin de créer une structure correcte de données topologiques en triant les points communs et des bords coïncidents.

Plus tard, vous verrez comment tester et examiner les données de

maillage.

Modélisation

Pour créer des formes géométriques régulières, vous pouvez utiliser le script Python **BuildRegularGeoms.py**.

```
import BuildRegularGeoms
```

Ce script fournit les méthodes pour construire des figures simples qui ont besoin d'une rotation comme des sphères, ellipsoïdes, cylindres, tores et cônes.
Et il existe aussi une méthode pour créer un simple cube.
Pour créer un tore, par exemple, nous ferons:

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

Les deux premiers paramètres définissent les rayons du tore, et le troisième paramètre est un facteur de sous-échantillonnage pour le nombre de triangles qui seront créés. Plus cette valeur est élevée plus la figure sera lisse et plus cette valeur est basse plus grossière sera la figure.
La classe Mesh offre un ensemble de fonctions booléennes qui peuvent êtres utilisées à des fins de modélisation. Il fournit l'union, l'intersection et la différence entre deux objets maillés.

```
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
```

Et ici, un exemple complet qui calcule l'intersection entre une sphère et un cylindre qui coupe la sphère.

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

```
d.addObject("Mesh::Feature","Diff_Sphere_Cylinder").Mesh=diff
d.recompute()
```

Examens et Test

Ecrire vos propres algorithmes

Exporter

Vous pouvez même écrire votre modèle de maillage dans un module Python:

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

Relations avec Gui (Interface graphique)

Modules supplémentaires à tester

Une extension (difficile à utiliser) de **scripts Mesh** qui est à tester.
Dans cette compilation test, toutes les méthodes sont appelées et toutes les propriétés et attributs sont manipulés.
Donc si vous êtes assez audacieux pour le tester, allez voir (http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Mesh/App/MeshTestsApp.py?view=markup) cette compilation de modules "unifié".

See also Mesh API

< précédent: FreeCAD Scripting Basics Index suivant: Topological data scripting >

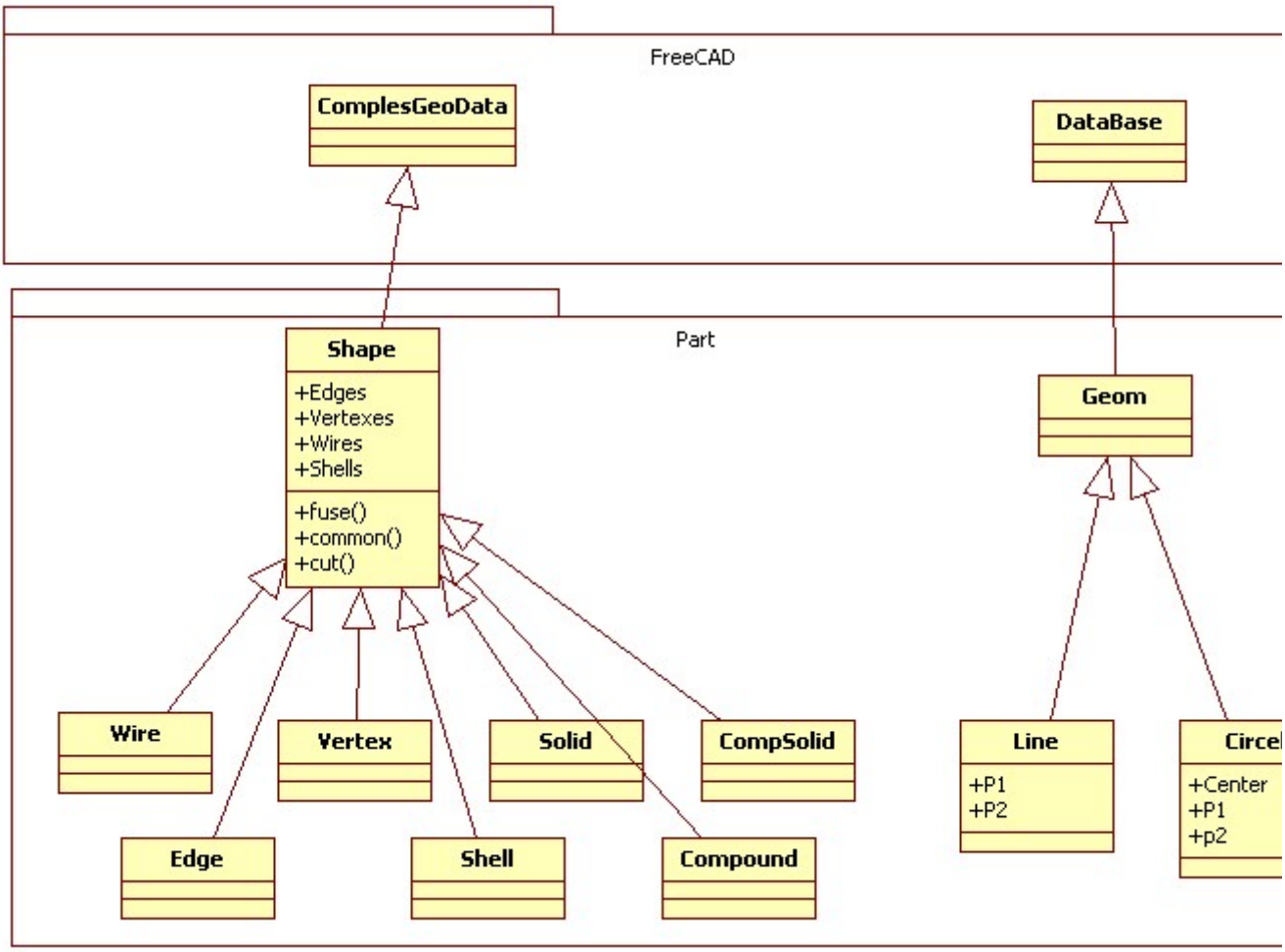
Cette page décrit différentes méthodes pour créer et modifier des pièces avec Python.
Avant de lire cette page, si vous n'êtes pas familier avec la programmation Python, vous pouvez vous diriger sur cette page d'introduction à Python et scripts de base en Python pour FreeCAD.

Introduction

Nous allons ici vous expliquer comment contrôler la **boîte à outils** (Part Module) ou de n'importe quel script externe, directement à partir de l'interpréteur Python inclus dans FreeCAD, .
Assurez-vous de parcourir l'article de familiarisation et scripts de base si vous avez besoin de plus amples renseignements sur la façon dont les scripts Python fonctionnent dans FreeCAD.

Class Diagram

Ceci est un Unified Modeling Language (UML) (http://fr.wikipedia.org/wiki/Unified_Modeling_Language) de la classe la plus importante de 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

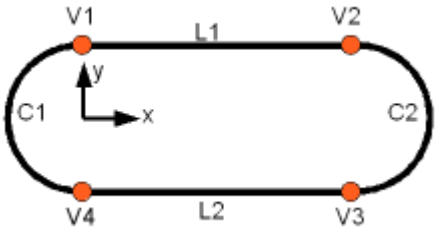
Sont aussi disponibles des données de type topologique:

- **Compound** Groupe de types différents d'objets topologiques.

- **Compsolid** Un groupe de solides reliés par leurs faces. C'est un concept des notions de **WIRE** (filaire,bord..) et **SHELL** (coquille,enveloppe) des solides.
- **Solid** Une portion de l'espace limité par son enveloppe. Il est en 3 dimensions.
- **Shell** Un groupe de faces reliés par leurs bords.Un "SHELL" peut être ouvert ou fermé.
- **Face** En 2D, c'est une surface plane; en 3D, c'est une seule face du volume. Sa géométrie est coupée par des contours. Il est en deux dimensions.
- **Wire** Un ensemble relié par ses VERTEX (sommets). Il peut être de contour ouvert ou fermé suivant si les sommets sont reliés ou non.
- **Edge** Elément topologique correspondant à une courbe retenue. Un "Edge" est généralement limité par des sommets. Il a une dimension.
- **Vertex** Elément topologiques correspondant à un point. Il n'a pas de dimension.
- **Shape** Est le terme générique pour traduire tout ce qui précède.

Exemple rapide : Création topologique simple

Nous allons créer une topologie avec une géométrie toute simple.
Nous devrons veiller à ce que les sommets des pièces géométriques soient à la même position, quatre sommets, deux cercles et deux lignes.



Création de la géométrie

Nous devons d'abord créer les parties distinctes géométriques en filaire.
Nous devons veiller à ce que tous les sommets des pièces géométriques qui vont êtres raccordées soient à la même position.
Sinon, plus tard nous pourrions ne pas être en mesure de relier

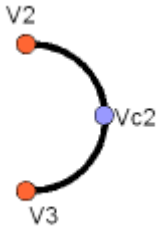
les pièces géométriques en une topologie!

Donc, nous créons d'abord les 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

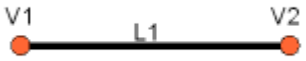
Pour créer un arc de cercle, nous créons un point de repère puis nous créons l'arc de cercle passant par trois points:




```
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)
```

Ligne

La ligne peut être créée très simplement en dehors des points :




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

Tout relier

La dernière étape consiste à relier les éléments géométriquement ensemble, et façonner une forme topologique:

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

Construire un prisme

Maintenant nous allons extruder notre forme filaire dans une direction, et créer une forme en 3 Dimensions:

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

Affichons le tout

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

Création de formes simples

Vous pouvez créer facilement des formes topologiques avec "**make...()**" qui est une méthode du "**Module Part**":

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

La combinaison de **make...()** avec d'autres methodes sont disponibles:

- **makeBox(l,w,h)**: Construit un cube et pointe sur **p** dans la direction **d** et de dimensions (**longueur,largeur,hauteur**).
- **makeCircle(radius)**: Construit un cercle de rayon (**r**).
- **makeCone(radius1,radius2,height)**: Construit un cône de (**rayon1,rayon2,hauteur**).
- **makeCylinder(radius,height)**: Construit un cylindre de (**rayon,hauteur**).
- **makeLine((x1,y1,z1),(x2,y2,z2))**: Construit une ligne aux coordonnées (**x1,y1,z1**),(**x2,y2,z2**) dans l'espace 3D.
- **makePlane(length,width)**: Construit un rectangle de (**longueur,largeur**).
- **makePolygon(list)**: Construit un polygone (**liste de points**).
- **makeSphere(radius)**: Construit une sphère de (**rayon**).
- **makeTorus(radius1,radius2)**: Construit un tore de (**rayon1,rayon2**).

La liste complète des API du module est sur la page Part API.

Importer les modules nécessaires

Nous avons d'abord besoin d'importer le module **Part** afin que nous puissions utiliser son contenu Python.
Nous allons également importer le module **Base** à l'intérieur du module de FreeCAD:

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

Création d'un Vecteur

Les Vecteurs (http://fr.wikipedia.org/wiki/Vecteur) sont l'une des informations les plus importantes lors de la construction des formes géométriques.
Ils contiennent habituellement 3 nombres (mais pas toujours) les coordonnées cartésiennes **x, y et z**.
Vous pouvez créez un vecteur comme ceci:

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

Nous venons de créer un vecteur de coordonnées **x = 3, y = 2, z = 0**.
Dans le module Part, les vecteurs sont utilisés partout.
Le module Part utilise aussi une autre façon de représenter un point, appelé Vertex, qui n'est actuellement rien d'autre qu'un conteneur pour un vecteur.
Vous pouvez accéder aux vecteurs d'un sommet comme ceci:

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

Création d'une arête (edge)

Une arête (bord) n'est rien d'autre qu'une ligne avec deux Vertex (sommets):

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

PS: Vous pouvez aussi créer un arête en donnant deux Vecteurs:

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

Vous pouvez trouver la longueur et le centre d'une arête comme ceci:

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

Mise en forme à l'écran

Jusqu'à présent, nous avons créé un objet a arêtes vives (bords), mais il n'est pas visible à l'écran.
C'est parce que nous n'avons manipulé que des objets en Python. L'écran FreeCAD n'affiche uniquement que les vues 3D que vous lui demandez d'afficher.
Pour cela, nous utilisons une méthode simple:

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

Un Objet 3D sera affiché dans notre document FreeCAD, et notre dessin sera affiché sous forme filaire.
Utilisez cette commande chaque fois que vous voudrez afficher votre forme géométrique à l'écran.

Création d'un contour (Wire)

Un contour est une ligne multi-arêtes, et peut être créé dans une liste d'arêtes ou même une liste de lignes (fils):

```
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 0190A
Part.show(wire3)
```

Part.show (wire3) permet d'afficher les 4 bords qui composent notre contour filaire.
D'autres informations utiles, peuvent être facilement récupérées:

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

Création d'une face

Seul les faces à contour fermés seront valides.
Dans cet exemple, wire3 est un contour fermé, et Wire2 est un contour ouvert (voir ci-dessus)

```
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
```

Seul les faces auront une superficie, mais les lignes et les bords (arêtes) n'en possède pas .

Création d'un cercle

Un cercle est créé simplement comme ceci:

```
circle = Part.makeCircle(10)
```

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

Si vous voulez le créer à une coordonnée précise, faites comme ceci:

```
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 sera créé à une distance de 10 à partir de l'axe d'origine x et sera orienté dans la direction de l'axe x.
Remarque: makeCircle accepte uniquement Base.Vector() pour la position mais pas les tuples (http://fr.wikipedia.org /wiki/Modèle_relationnel) normaux.
Vous pouvez également créer un arc de cercle en donnant l'angle de départ et l'angle de la fin comme suit:

```
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)
```

Si nous joignions les deux arcs **arc1** et **arc2** nous obtiendrons un cercle.
L'angle fourni doit être exprimé en degrés, s'il sont en radians, vous devez les convertir en degrès avec la formule: **degrés = radians * 180/PI** ou en utilisant le module mathématiques Python (après avoir fait **import math**, bien sûr):

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

Création d'un arc sur des points (repères)

Malheureusement, il n'existe pas de fonction **makeArc** mais nous avons la fonction **Part.Arc** pour créer un arc sur trois points de référence.
Fondamentalement, nous pouvons supposer un arc attaché sur un **point de départ**, passant sur un **point central** et se termine sur un **point final** en .
Part.Arc crée un objet **arc** pour lequel **.ToShape()** doit être appelée pour obtenir un objet ligne (edge), de cette manière nous

utiliserons **Part.Line** lieu de **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 travaille uniquement avec **Base.Vector()** pour les points mais pas pour les tuples.
arc_edge est ce qui sera affiché à l'aide **Part.show** (arc_edge).
Vous pouvez également obtenir un arc de cercle en utilisant une partie de cercle:

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

Les arcs Arc sont des lignes (edges). Ils peuvent donc, être utilisés aussi comme contour en filaire.

Création de polygones

Un polygone est tout simplement une ligne (wire) avec de multiples lignes droites.
La fonction **makePolygon** crée une liste de points et crée une ligne de points en points:

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

Création de courbes de Bézier

Les courbes de Bézier (http://fr.wikipedia.org/wiki/Courbe_de_B%C3%A9zier) sont utilisées pour modéliser des courbes lisses à l'aide d'une série de repères (points de contrôle) avec un nombre de repères représentants la précision (fluidité de la courbe) optionnel. La fonction ci-dessous fait un **Part.BezierCurve** avec une série de points FreeCAD.Vector. (Note : l'indice du premier repère et du nombre **commencent à 1**, et pas à 0.)

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

```
    edge = Part.Edge(geomCurve)
    return(edge)
```

Création d'une forme plane

Une forme plane, est tout simplement une surface plane rectangulaire.
La méthode utilisée pour créer une forme plane est la suivante: **makePlane(longueur, largeur, [point de départ, direction])**.
Par défaut **point de départ** = Vecteur(**0,0,0**) et **direction** = Vecteur(**0,0,1**).
L'utilisation **point de départ = Vecteur(0,0,1)** va créer la forme sur le plan **axe z**, tandis que **direction = Vecteur(1,0,0)** va créer la forme sur le plan **axe x**:
(Pour s'y retrouver un peu sur les axes, Vecteur (0 , 0 , 1) *est égal à* ***Vecteur (X=0 , Y=0 , Z=1) l'ordre des axes sera toujours (x , y , z))***

```
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.BoundBox
> BoundBox (3, 0, 0, 5, 0, 2)
```

BoundBox est un rectangle qui possède une diagonale commençant sur le plan (3,0,0) et se terminant à (5,0,2).
L'épaisseur de la boîte (Box) dans l'axe y est égal à zéro, car notre forme est totalement plane.

PS: makePlane accepte uniquement **Base.Vector()** pour **start_pnt** et **dir_normal** mais ici, pas de tuples

Création d'une ellipse

Pour créer une ellipse, il existe plusieurs façons:

```
Part.Ellipse()
```

Créez une ellipse avec, grand rayon = 2, petit rayon = 1 et centre = (0,0,0)

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

Créez une copie des données de l'ellipse

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

Crée une ellipse positionnée au point "**Center**", le plan de l'ellipse est défini par **Center**, **S1** et **S2**,
le **grand axe** est définit par **Center** et **S1**,
son grand rayon est la distance entre **Center** et **S1**,
son petit rayon est la distance entre **S2** et le **grand axe**.

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

Crée une ellipse avec un grand rayon **MajorRadius** et un petit rayon **MinorRadius**, et situé dans le plan défini par **(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())
```

Dans le code ci-dessus, nous avons passé **S1** (Grand rayon), **S2** (Petit rayon) et le **centre** (les coordonnées centrales).
De même que l'**Arc**, l'**Ellipse** crée également un objet **Ellipse** mais pas d'arête (bords), nous avons donc besoin de le convertir en arête à l'aide **toShape()** pour l'afficher.

PS: **Arc** accepte uniquement **Base.Vector()** pour les **points** mais pas les **tuples**.

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

pour construire l'**Ellipse** ci-dessus, nous avons entré les coordonnées **centrales**, le **Grand rayon** et le **Petit rayon**.

Création d'un Tore

Nous créons un Tore en utilisant la méthode **makeTorus(rayon1 , rayon2 , [pnt , dir , angle1 , angle2 , angle])**.

Par défaut,
Rayon1 = est le rayon du grande cercle
Rayon2 = est le rayon du petit cercle,
pnt = Vecteur(0,0,0),**pnt** est le centre de tore
dir = Vecteur(0,0,1), **dir** est la direction normale
angle1 = 0, est l'angle de début pour le petit cercle exprimé en radians
angle2 = 360 est l'angle de fin pour le petit cercle exprimé en radians
angle = 360 le dernier paramètre est la section du tore

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

Le code ci-dessus créera un **tore** avec un diamètre de 20 (rayon de 10) et une épaisseur de 4 (rayon du petite cerlce 2)

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

Le code ci-dessus créera une portion du tore

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

Le code ci-dessus créera un demi tore, seul le dernier paramètre change à savoir l'angle et, les angles restants sont prédéfinis.
En donnant un angle de 180 degrés, crée un tore de 0 à 180 degrés, c'est à dire un demi tore.

Création d'un cube ou d'un parallélépipède

Utilisez **makeBox (longueur , largeur , hauteur , [pnt , dir])**.
Par défaut **pnt=Vector(0,0,0)** and **dir=Vector(0,0,1)**

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

Création d'une Sphère

Nous utiliserons **makeSphere (rayon , [pnt , dir , angle1 , angle2 , angle3])**.
rayon = rayon de la sphère par défaut,
pnt = Vecteur (0,0,0),
dir = Vecteur (0,0,1),
angle1 = -90, verticale minimale de la sphère
angle2 = 90, verticale maximale de la sphère
angle3 = 360, le diamètre de la sphère elle-même

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

Création d'un Cylindre

Nous utiliserons **makeCylinder (radius , height , [pnt , dir ,angle])**.
Par défaut,
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)
```

Création d'un Cône

Nous utiliserons **makeCone (radius1 , radius2 , height , [pnt , dir , angle])**.
Par défaut,
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)
```

Modification d'une forme

Il ya plusieurs manières de modifier des formes. Certaines sont de simples opérations de transformation telles que le déplacement ou la rotation de formes, d'autres, sont plus complexes, tels que fusion et en soustraction d'une forme à une

autre. Tenez en compte.

Opérations de Transformation

Transformer une forme

La transformation est l'action de déplacer une forme d'un endroit à un autre.
Toute forme (arête, face, cube, etc ..) peut être transformé de la même manière:

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

Cette commande va déplacer notre forme "**myShape**" de 2 unités dans la direction x.

Rotation d'une forme

Pour faire pivoter une forme, vous devez spécifier le centre de rotation, l'axe, et l'angle de rotation:

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

Cette opération va faire pivoter notre forme de 180 degrés sur l'axe z.

Transformations génériques avec matrices

Une matrice est un moyen très simple de mémoriser les transformations dans le mode 3D. Dans une seule matrice, vous pouvez définir les valeurs de transformation, rotation et mise à l'échelle à appliquer à un objet.
Par exemple:

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

PS: les matrices de FreeCAD travaillent en radians. En outre, presque toutes les opérations matricielles qui travaillent avec un vecteur peut aussi avoir 3 nombres, de sorte que ces 2 lignes effectuent le même travail:

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

Lorsque notre matrice est paramétrée, nous pouvons l'appliquer à notre forme. FreeCAD fournit nous fournit 2 méthodes: **transformShape()** et **transformGeometry()**.
La différence est que, avec la première, vous ne verez pas de différence (voir "**mise à l'échelle d'une forme**" ci-dessous).
Donc, nous pouvons opérer notre transformation comme ceci:

```
myShape.trasformShape(myMat)
```

ou

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

Echelle du dessin (forme)

Changer l'échelle d'une forme est une opération plus dangereuse, car, contrairement à la translation ou à la rotation, le changement d'échelle non uniforme (avec des valeurs différentes pour x, y et z) peut modifier la structure de la forme!
Par exemple, le redimensionnement d'un cercle avec une valeur plus élevée horizontalement que verticalement le transformera en une ellipse, qui mathématiquement très différent.
Pour modifier l'échelle, nous ne pouvons pas utiliser le transformShape, nous devons utiliser **transformGeometry()**:

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

Opérations Booléennes

Soustraction

Soustraire une forme d'une autre est appelé, dans le jargon OCC (http://www.opencascade.org/org/doc/)/FreeCAD "**cut**" (coupe) et, se fait de cette manière:

```
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

De la même manière, l'intersection entre 2 formes est appelé "**common**" et se fait de cette manière:

```
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)
```

Fusion

La fusion "**fuse**", fonctionne de la même manière:

```
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

Une section, est l'intersection entre une **forme solide** et une **forme plane**.
Il retournera une courbe d'intersection, et sera composé de bords (edges, arêtes)

```
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

L'extrusion est une action de "pousser" une forme plane dans une certaine direction et résultant en un corps solide.
Par exemple, pousser sur un cercle pour le transformer en tube:

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

Si votre cercle est vide, vous obtiendrez un tube vide.
Mais si votre cercle est un disque, avec une face pleine, vous obtiendrez un cylindre solide:

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

Exploration de la forme (shape)

Vous pouvez facilement explorer la structure de ses données topologique:

```
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
```

En tapant ce code dans l'interpréteur Python, vous aurez une bonne compréhension de la structure de **Part objets**.
Ici, notre commande **makebox()** créé une forme solide. Ce solide, comme tous les solides **Part**, contiennent des **faces**. Une **face** est constituée de **lignes**, qui sont un ensemble de **bords, arêtes** qui délimitent la face. Chaque face a au moins un contour fermé (il peut en avoir plus si la face comporte un ou plusieurs trou). Dans une ligne, nous pouvons voir chaque côté séparément, et nous pouvons voir les sommets (Vertex) de

chaque bord ou arête. Lignes et arêtes n'ont que deux sommets, évidemment.

Analyse des arêtes (Edge)

Dans le cas d'un bord (ou arête), qui est une courbe arbitraire, il est fort probable que vous voulez faire une discrétisation. Dans FreeCAD, les bords sont paramétrés par leurs longueurs.
Cela signifie, que vous pouvez suivre une arête/courbe par sa longueur:

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

Maintenant, vous pouvez accéder à un grand nombre de propriétés de l'arête en utilisant sa longueur comme une position. C'est à dire que, si l'arête(ou bord) a une longueur de 100 mm la position de départ est **0** et sa position extrème est **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)
```

Utilisation de la sélection

Ici, nous allons voir comment nous pouvons utiliser la fonction de sélection, quand l'utilisateur a fait une sélection dans la visionneuse.
Tout d'abord, nous créons une boîte (box), et nous le voyons et la sélectionnons dans la visionneuse.

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

Sélectionnez maintenant quelques faces ou arêtes.
Avec ce script, vous pouvez parcourir tous les objets sélectionnés

et visionner leurs sous-éléments:

```
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
```

Sélectionnez quelques bords et ce script va calculer la longueur:

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

Exemple Complet: "The OCC bottle"

Un exemple typique, trouvée sur OpenCasCade Getting Started Page (http://www.opencascade.org/org/gettingstarted/appli/) vous montre comment construire une bouteille.
C'est un excellent exercice pour FreeCAD. En fait, vous pouvez suivre notre exemple ci-dessous et regarder simultanément la page OCC (http://www.opencascade.org/org/doc/), vous comprendrez comment les structures **OCC** sont misent en œuvre dans FreeCAD.
Le script complet ci-dessous de **MakeBottle.py** est également inclus dans l'installation de FreeCAD dans le dossier **Mod/Part** et peut être appelé à partir de l'interpréteur Python en tapant:

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

Le script complet

Ici, le script complet de **MakeBottle.py** (extension .py):

```
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
```

Détail et déroulement MakeBottle.py

```
import Part, FreeCAD, math
from FreeCAD import Base
```

Nous aurons besoin, bien sûr, du module **Part**, mais aussi du module **FreeCAD.Base**,
qui contient les structures de base de FreeCAD comme les **vectors** et **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)
```

Ici, nous définissons notre fonction MakeBottle.
Cette fonction peut être appelée sans argument, comme nous l'avons fait ci-dessus, les valeurs par défaut, de largeur, hauteur et épaisseur seront utilisés.
Ensuite, nous définissons une paire de points qui seront utilisés pour la construction de notre profil de base.

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

C'est ici que nous définissons les formes géométriques: un arc, composé de 3 points, et deux segments de ligne, de 2 points chacun.

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

Rappelez-vous la différence entre la **géométrie** et les **formes**? Nous allons construire les formes de notre forme géométrique. 3 bords (bords ou arêtes peuvent être des segments de droites ou des courbes), puis nous raccordons tous les sommets.

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

Jusqu'à présent, nous n'avons construit que la moitié du profil. Qui est plus facile que de construire l'ensemble du profil, et nous allons simplement refléter l'autre moitié du profil, et coller les deux moitiés ensemble. Nous allons donc d'abord créer une matrice. Une matrice, est un mode opératoire pour appliquer des transformations aux objets dans le monde de la 3D, car, il peut contenir dans une seule structure toutes les transformations de base qui peuvent êtres fait sur les objets 3D (déplacement, rotation et échelle). Nous créons la matrice, nous lui faisons subir un effet miroir, et nous créons une copie de notre dessin avec cette matrice. C'est de cette façon, que la transformation est appliquée. Nous avons maintenant deux contours, et nous pouvons avec eux faire un troisième contours, les contours sont

en fait des listes de bords.

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

Maintenant, nous avons un contour fermé, il peut être transformé en une face. Une fois que nous avons une face, nous pouvons l'extruder.
Une fois fait, nous avons un solide. Puis, nous appliquons arrondi à notre objet, car nous voulons lui donner un aspect "design", n'est-ce pas?

```
   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)
```

Maintenant, le corps de la bouteille est fait, nous avons encore besoin de créer le goulot.
Donc, nous construisons un nouveau solide, avec un cylindre.

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

L'opération de fusion, qui dans d'autres applications est parfois appelé union, est très puissante.
Cette opération prendra soin de coller ce qui doit être collé et enlever ce qui doit être enlevé.

```
   return myBody
```

Puis, nous revenons à notre bouteille (Part solid), qui est le résultat de notre fonction (**return myBody**).
Ce Part solid, comme toute autre forme solide, peut être attribuée à un Objet dans un document FreeCAD, avec:

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

ou, encore plus simple:

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

Cube percé

Ici un exemple complet de construction d'un cube percé.

La construction se fait face par face et quand le cube est terminé, il est évidé d'un cylindre traversant.

```
import Draft, Part, FreeCAD, math, PartGui, FreeCADGui, PyQt4
from math import sqrt, pi, sin, cos, asin
from FreeCAD import Base

size = 10
poly = Part.makePolygon( [ (0,0,0), (size, 0, 0), (size, 0, size), (0, 0, size), (0, 0, 0)])

face1 = Part.Face(poly)
face2 = Part.Face(poly)
face3 = Part.Face(poly)
face4 = Part.Face(poly)
face5 = Part.Face(poly)
face6 = Part.Face(poly)

myMat = FreeCAD.Matrix()
myMat.rotateZ(math.pi/2)
face2.transformShape(myMat)
face2.translate(FreeCAD.Vector(size, 0, 0))

myMat.rotateZ(math.pi/2)
face3.transformShape(myMat)
face3.translate(FreeCAD.Vector(size, size, 0))

myMat.rotateZ(math.pi/2)
face4.transformShape(myMat)
face4.translate(FreeCAD.Vector(0, size, 0))

myMat = FreeCAD.Matrix()
myMat.rotateX(-math.pi/2)
face5.transformShape(myMat)

face6.transformShape(myMat)
face6.translate(FreeCAD.Vector(0,0,size))

myShell = Part.makeShell([face1,face2,face3,face4,face5,face6])

mySolid = Part.makeSolid(myShell)
mySolidRev = mySolid.copy()
mySolidRev.reverse()

myCyl = Part.makeCylinder(2,20)
myCyl.translate(FreeCAD.Vector(size/2, size/2, 0))

cut_part = mySolidRev.cut(myCyl)

Part.show(cut_part)
```

Chargement et sauvegarde

Il ya plusieurs façons de sauver votre travail dans le Part Module

. Vous pouvez bien sûr sauvegarder votre document au format FreeCAD, mais vous pouvez également enregistrer les objets directement dans un format courant de CAO, tels que BREP (http://fr.wikipedia.org/wiki/B-Rep), IGS (http://fr.wikipedia.org/wiki/Initial_Graphics_Exchange_Specification), STEP (http://en.wikipedia.org/wiki/Step_(software)) et STL (http://fr.wikipedia.org/wiki/STL_(format)).

L'enregistrement d'une forme (un projet) dans un fichier est facile, il y a les fonctions **exportBrep()**, **exportIges()**, **exportStl()** et **exportStep()** qui sont des méthodes disponibles pour toutes les formes d'objets.
Donc, en faisant:

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

Ceci sauve votre box (cube) dans le format **.STP**
Pour ouvrir un fichier BREP, IGES ou STEP simplement en faisant le contraire:

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

Pour convertir un fichier .stp en .igs faites simplement :

```
import Part
s = Part.Shape()
s.read("file.stp")       # incoming file igs, stp, stl, brep
s.exportIges("file.igs") # outbound file igs
```

Notez que l'importation ou l'ouverture de fichiers **BREP**, **IGES** ou **STEP** peut également être effectuée directement à partir du **Menu Fichier -> Ouvrir**, **Menu Fichier -> Importer** ou l'icone "**Ouvrir un document ou importer des fichiers**", et pour l'exportation d'un fichier par **Menu Fichier -> Exporter**

< précédent: Mesh Scripting/fr Index suivant: Mesh to Part/fr >

Converting Part objects to Meshes/fr

La conversion des objets de haut niveau tels que les objets (formes) en objets simples comme les mailles (Mesh) est une opération facile, où, toutes les faces d'un Objet Part deviennent une composition de triangles (exemple sur le site de coin3d un des moteurs de FreeCAD) (http://www.coin3d.org/usage/casestudies/users/usageexample.2008-05-30.6001136448/4DVista.PNG).
Le résultat de cette triangulation (tessellation (http://en.wikipedia.org/wiki/Tessellation)) est ensuite utilisé pour construire un maillage (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)
```

Parfois, la triangulation de certaines faces offertes par OpenCascade (http://www.opencascade.org/) sont assez laid. Si une face a un forme rectangulaire et ne contient pas de trous ou n'est pas limité par des courbes, vous pouvez également créer un maillage sur cette forme:

```
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
```

Conversion de Mailles en Part objects

La conversion des mailles en Part objets est une opération extrêmement importante en CAO, car, très souvent vous recevrez des données 3D au format Mesh (maillage) à partir d'autres utilisateurs ou émis par d'autres applications de CAO. Les Mailles sont très pratiques pour représenter les formes géométriques libres et de grandes scènes visuelles, car il est très léger, mais pour la CAO nous préférons généralement des objets de niveau supérieur qui portent beaucoup plus d'informations comme, l'idée de solides, ou faces sont faites de courbes au lieu de triangles.

La conversion des mailles en un de ces objets de niveau supérieur (gérée par le Part Module dans FreeCAD) n'est pas une opération facile. Les Mailles peuvent êtres faites de milliers de triangles (par exemple lorsqu'ils sont générés par un scanner 3D), et des solides faits du même nombre de faces serait extrêmement lourd à manipuler. Donc, vous voudrez généralement voir l'objet optimisé lors de la conversion.

FreeCAD propose actuellement deux méthodes pour convertir des Parts objets en mailles. La première méthode est simple, la conversion directe, sans aucune optimisation:

```
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)
```

La seconde méthode, offre la possibilité d'examiner les aspects de mailles coplanaires, lorsque l'angle entre eux est sous une certaine valeur. Cela permet de construire des formes beaucoup plus simples:

```
# 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.BoundBox.DiagonalLength > max_length:
             max_length = i.BoundBox.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 lsit
       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)
```

< précédent: Topological data scripting suivant: Scenegraph >
Index

De base, FreeCAD est une puissante compilation de différentes bibliothèques graphiques, la plus importante étant OpenCascade (http://en.wikipedia.org/wiki/Open_CASCADE), pour la gestion et la construction des formes géométriques, Coin3d (http://en.wikipedia.org/wiki/Coin3D) pour l'affichage des formes géométriques, et Qt (http://fr.wikipedia.org/wiki/Qt) pour créer une interface utilisateur graphique (GUI) agréable et fonctionnelle.

Les formes géométriques qui apparaissent dans les vues 3D de FreeCAD sont des rendus obtenus par la bibliothèque Coin3D (Coin3D est une application de OpenInventor standard (http://fr.wikipedia.org/wiki/Inventor_(bibliothèque_logicielle))).

Le logiciel OpenCascade fournit les même fonctionnalités que coin3D, mais, dans les débuts de FreeCAD, il a été décidé de ne pas utiliser le moteur d'OpenCascade et de se tourner plutôt vers le logiciel coin3D plus performant. Une bonne façon de se renseigner sur cette bibliothèque est de lire le livre Open Inventor Mentor (http://www-evasion.imag.fr/Membres/Francois.Faure/doc/inventorMentor/sgi_html/).

Actuellement OpenInventor (http://fr.wikipedia.org/wiki/Inventor_(bibliothèque_logicielle)) est un langage de description de scènes en 3 dimensions. La scène décrite dans OpenInventor est restituée en OpenGL sur votre moniteur. Coin3D prend en charge toutes ces procédures, de telle sorte que le programmeur n'a pas besoin de traiter les appels complexes d'OpenGL, il lui suffit simplement de fournir le code OpenInventor adéquat.

Le gros avantage d'OpenInventor est, qu'il est une norme fort bien connue et très bien documentée.
Le gros travail que FreeCAD fait pour vous, consiste essentiellement à traduire les informations sur les formes géométriques OpenCascade en langage OpenInventor.

OpenInventor décrit une scène 3D sous la forme d'une scène graphique (http://fr.wikipedia.org/wiki/Graphe_de_scène) , comme le montre l'exemple ci dessous:

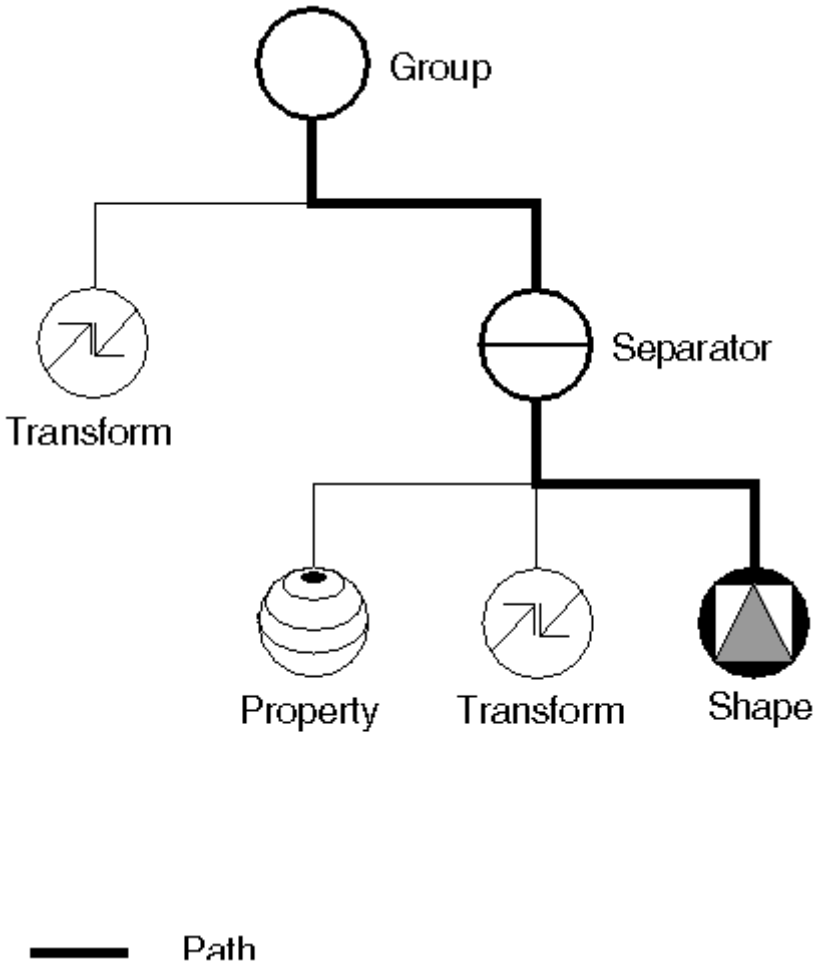


image from Inventor mentor (http://www-evasion.imag.fr/~Francois.Faure/doc/inventorMentor/sgi_html/index.html)

OpenInventor scenegraph, décrit tout ce qui fait partie d'une scène 3D, comme les formes géométriques, les couleurs, les matériaux, les lumières etc., et structure toutes les données d'une manière claire et précise.
Cette structure peut être groupée en sous-structures vous permettant d'organiser le contenu de votre scène de la manière qui vous conviens le mieux.
Voici un exemple d'un fichier OpenInventor:

```
#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 {
      }
   }
}
```

Comme vous pouvez le voir, la structure est très simple. Vous utilisez des séparateurs **(Separator)** pour organiser vos blocs de données, un peu comme vous le feriez pour organiser vos fichiers dans des dossiers.
Chaque instruction influe celle qui suit, par exemple, les deux premiers articles à la racine de nos **Separator** sont une rotation (***RotationXYZ {..}***) et une transformation (***Transform {..}***), ils auront une incidence directe sur tous les éléments suivants (*comme, si vous changez l'attribut d'un dossier, tous les sous dossiers seront affectés*).
Dans un séparateur, nous définirons la matière, dans un autre, la transformation. Notre cylindre sera donc affecté par les deux transformations, celle qui lui a été appliqué directement et celle qui a été appliquée à son séparateur parent (**Separator{..Separator{..}}** à la manière des dossiers dans un disque dur).

Nous avons également beaucoup d'autres d'éléments pour organiser notre scène (projet), tels que des groupes, des commutateurs ou des annotations.
Nous pouvons donner à nos objets des définitions très complexes, de la couleur, des textures des modes d'ombrage et de transparence. Nous pouvons aussi définir de la lumière, des caméras et, même du mouvement.
Il est aussi possible d'intégrer des portions de scripts dans des fichiers OpenInventor et de définir des comportements plus complexes.

Si vous voulez en apprendre plus sur OpenInventor, allez tout de suite sur The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor (http://www-evasion.imag.fr /~Francois.Faure/doc/inventorMentor/sgi_html/index.html).

Normalement, dans FreeCAD, nous n'avons pas besoin d'interagir

directement avec scenegraph OpenInventor.
Dans un document FreeCAD, chaque objet maillage, forme de la pièce ou toute autre chose, est automatiquement converti en code OpenInventor et est inséré dans la scène graphique que vous voyez dans la vue 3D.

Toutes modifications dans le document, ajout ou suppression d'objets, sont en permanence mises à jour dans la scène graphique. En fait, chaque objet (dans l'espace de l'Application), dispose d'un constructeur de la vue (un objet correspondant dans l'espace Gui), responsable de la création du code OpenInventor.

Mais il y a de nombreux avantages à pouvoir accéder directement à la scène graphique. Par exemple, nous pouvons modifier temporairement l'apparence d'un objet, ou nous pouvons ajouter des objets à la scène qui n'ont aucune existence réelle dans le document FreeCAD, tels que la construction de la géométrie, les aides, conseils graphiques ou des outils qui permettent des manipulations ou des informations à l'écran.

FreeCAD dispose de plusieurs outils pour voir ou modifier le code OpenInventor.

Par exemple, le code Python suivant, montre la représentation OpenInventor d'un objet sélectionné:

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

Mais nous avons aussi un module Python qui permet un accès complet à toute chose gérée par Coin3D, comme, notre scène graphique FreeCAD.
Alors, lisez la suite sur la page de pivy.

< précédent: Mesh to Part Index suivant: Pivy >

Pivy (http://pivy.coin3d.org/) est une bibliothèque de codes qui sert de passerelle entre Python et Coin3d (http://www.coin3d.org), la bibliothèque 3D de rendu graphique utilisé par FreeCAD. Lors de l'importation dans l'interpréteur Python, Pivy permet de dialoguer immédiatement avec les procédures de Coin3d, tels que les vue3D, ou même d'en créer de nouvelles.
Pivy est inclus d'origine dans l'installation FreeCAD.

La bibliothèque d'outils est divisée en plusieurs parties,

- **coin:** pour manipuler formes graphiques (projet) et gérer le système graphique (GUI), tels que plusieurs fenêtres ou, dans notre cas,
- **qt :** pour les interfaces.

Ces modules sont aussi accessibles à pivy, s'ils sont présents sur le système bien sûr.
Le module **coin** est toujours présent, et de toute façon c'est lui que nous allons utiliser, nous n'aurons pas besoin de nous occuper de l'affichage 3D dans une interface, FreeCAD s'en occupe lui-même.
Tout ce que nous aurons à faire, c'est:

```
from pivy import coin
```

Accéder et modifier une scène graphique

Nous avons vu dans la page Scenegraph comment coin organise une scène. Tout ce qui est affiché en 3D dans FreeCAD est construit et géré par **coin**.
Nous avons une **racine**, et tous les objets sur l'écran sont ses **enfants**, reliés par des **nodes** (noeuds). Les enfants aussi peuvent avoir une descendance.

FreeCAD a un moyen facile d'accéder a la racine d'une scène 3D:

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

```
print sg
```

La racine de la scène 3D sera:

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

Vous pouvez inspecter immédiatement les enfants (branches) de la scène 3D:

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

Certains de ces nodes, comme **SoSeparators** ou **SoGroups**, peuvent avoir des enfants eux-mêmes. La liste complète des **coin objets** disponibles peut être trouvée dans la documentation official coin documentation (http://doc.coin3d.org /Coin/classes.html).

Maintenant essayons d'ajouter quelque chose à notre scène (projet).
Nous allons ajouter un beau cube rouge:

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

et voici notre (beau) cube rouge.
Maintenant, nous allons essayer ceci:

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

Vu ? Tout est encore accessible et modifiable à la volée.
Pas besoin de recalculer ou redessiner quoi que ce soit, **coin** s'occupe de tout. Vous pouvez ajouter ce que vous voulez à votre scène (projet), propriétés de changements, cacher des objets, montrer des objets temporairement, faire n'importe quoi.
Bien sûr, cela ne concerne que l'affichage de la vue 3D.
L'affichage du document ouvert est recalculé par FreeCAD, et

recalcule un objet quand il a besoin de l'être.
Donc, si vous changez l'aspect d'un objet existant dans FreeCAD, ces modifications seront perdues si l'objet est recalculé, ou lorsque vous rouvrez le fichier.

Un truc, pour travailler avec scenegraphs dans vos scripts, vous pouvez, lorsque c'est nécessaire accéder à certaines propriétés des nodes que vous avez ajoutés.
Par exemple, si nous voulions faire évoluer notre cube, nous aurions ajouté un **node SoTranslation** à notre node personnalisé et,
il aurait ressemblé à ceci:

```
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)
mtCustomNode.addChild(trans)
myCustomNode.addChild(cub)
sg.addChild(myCustomNode)
```

Souvenez-vous que dans une scène graphique **OpenInventor**, l'ordre est très important. Un noeud affecte ce qui suit, de sorte que, si vous dites: couleur rouge, cube, couleur jaune, sphère ! vous obtiendrez un cube rouge et une sphère jaune.
Si nous traduisons maintenant notre noeud personnalisé, il vient après le cube, et ne l'affecte pas.
Si nous l'avions inséré lors de sa création, comme l'exemple ci-dessus,
nous pourrions faire maintenant:

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

Et notre cube sauterait de 2 unités vers la droite.
Enfin, pour supprimer quelque chose, nous ferons:

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

Utilisation des mécanismes de rappel

(callback)

Un mécanisme de rappel (http://fr.wikipedia.org/wiki/Fonction_de_rappel) est un système qui permet à la bibliothèque que vous utilisez, comme notre bibliothèque **coin** de faire un rappel comme, rappeler une certaine fonction pour l'Objet Python en cours d'exécution.
Cela est extrêmement utile, car, de cette manière coin peut vous avertir si un événement particulier survient dans la scène.
Coin peut voir des choses très différentes, comme, la position de la souris, les clics sur un bouton de la souris, les touches du clavier qui sont pressées, et bien d'autres choses.

FreeCAD dispose d'un moyen facile pour utiliser ces rappels:

```
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()
```

Le rappel doit être initiée à partir d'un objet, et, cet objet doit **toujours** être actif au moment du rappel.
Voir aussi une liste complète des événements possibles et leurs paramètres (en), ou dans la documentation officielle de coin (http://doc.coin3d.org/Coin/classes.html).

Documentation

Malheureusement, pivy ne dispose pas encore d'une documentation appropriée, mais puisqu'il s'agit d'une traduction exacte de coin, vous pouvez utiliser en toute sécurité la documentation révérencielle de coin (http://doc.coin3d.org/Coin/classes.html), et utiliser le style Python au lieu du style c ++ (par exemple **SoFile::getClassTypeId()** en c++, serait **SoFile.getClassId()** en pivy)

< précédent: Scenegraph Index suivant: PySide >

PySide

PySide (http://fr.wikipedia.org/wiki/PySide)est un Python obligatoire de l'outil mutiplateforme GUI de Qt. FreeCAD utilise PySide pour tous les GUI (Interface Graphique Utilisateur). PySidea évolué à partir du package PyQt qui était auparavant utilisé par FreeCAD pour son interface graphique. Voir Differences Between PySide and PyQt (http://qt-project.org /wiki/Differences_Between_PySide_and_PyQt) pour plus d'information sur ces différences.

Les utilisateurs de FreeCAD atteingnent souvent les limites tout en utilisant l'interface intégrée. Mais pour les utilisateurs qui souhaitent personnaliser leurs opérations alors l'interface Python existe et est documentée dans le Didacticiel de scripts Python. L'interface Python pour FreeCAD avait une grande flexibilité et de la puissance.Pour cette interaction de l'utilisateur Python avec Freecad , on utilise PySide, qui est est documenté sur cette page.

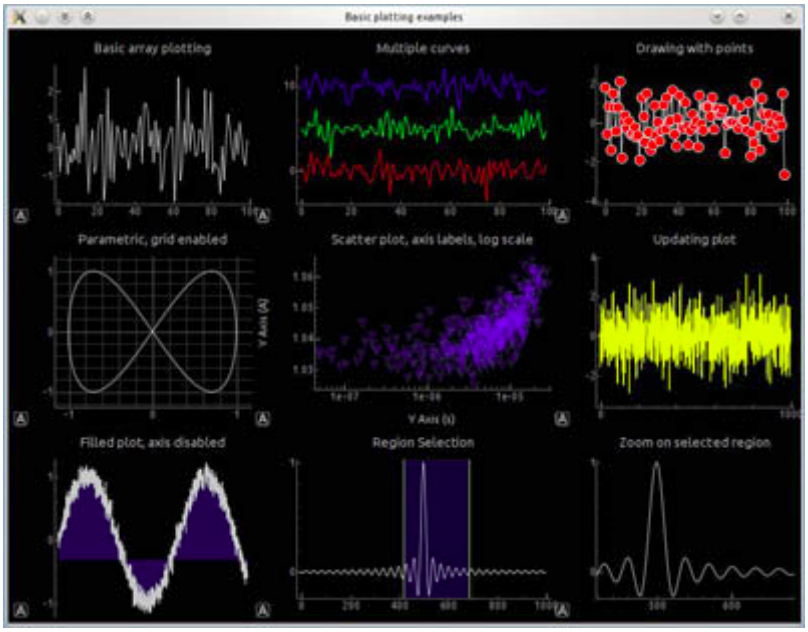
Python offre la mention «d'impression» qui donne le code:

```
print 'Hello World'
```

Avec l'instruction Python print vous avez seulement un contrôle limité de l'apparence et du comportement. PySide fournit le contrôle manquant et gère également les environnements (tels que l'environnement de fichier macro FreeCAD) où les installations intégrées de Python ne sont pas suffisantes

Les capacités de PySide varient de:

à



PySide est décrit dans les 3 pages suivantes qui doivent se suivre l'une après l'autre

- Exemples PySide Débutant (Bonjour tout le monde, annonces, saisir du texte, entrez le numéro)
- Exemples PySide intermédiaire (fenêtre dimensionnement, cacher des widgets, des menus contextuels, position de la souris, les événements de souris)
- Exemples PySide avancés (widgets, etc.)

Elles divisent l'objet en 3 parties, différenciées selon le niveau de connaissance de PySide, Python et l FreeCAD. La première page est un aperçu et un documents de référence donnant une description de PySide et comment il est mis en place tandis que les deuxième et troisième pages sont pour la plupart des exemples de code à différents niveaux.

L'intention est que les pages associées fourniront du code Python simple pour exécuter PySide de sorte que l'utilisateur travaillant sur un problème peut facilement copier le code, le collez-le dansson propre travail, l'adapter si nécessaire et retourner à leur résolution de problèmes avec FreeCAD. J'espère qu'ils n' auront pas à aller fouiller à travers l'Internet à la recherche de réponses aux questions PySide. Dans le même temps cette page n' est pas destinée à remplacer les différents tutoriels PySide complets et les sites de référence disponibles sur le web.

< previous: Pivy Index next: Scripted objects >

Outre les types d'objets standards tels que les annotations, les mailles et les objets Parts, FreeCAD offre également la possibilité incroyable d'écrire des scripts d'objets 100% Python. Ces "objets" se comporteront exactement comme n'importe quels autres objets dans FreeCAD, et sont, sauvegardés et restaurés automatiquement dans le répertoire de chargement/sauvegarde. Une particularité doit être comprise, ces objets sont enregistrés dans des fichiers **FreeCAD FcStd** avec le module Python **cPickle (http://docs.python.org/release/2.5/lib/module-cPickle.html)**. Ce module transforme un objet (code) Python en une chaîne de caractères (texte), lui permettant d'être ajouté au fichier sauvegardé.

Une fois chargé, le module **cPickle** utilise cette chaîne pour recréer l'objet d'origine, à condition qu'il ait accès au code source qui l'a créé.

Cela signifie que si vous enregistrez un tel objet personnalisé et l'ouvrez sur une machine où le code source Python qui a créé l'objet n'est pas présent, l'objet ne sera pas recréé.
Si vous distribuez ces scripts à d'autres, vous devrez aussi distribuer l'ensemble du script Python qui l'a créé.

Les fonctionnalités de Python suivent les mêmes règles que toutes les fonctionnalités de FreeCAD: ils sont séparés en plusieurs parties celle **App (application)** et **GUI parts (interface graphique)**.
La partie **Object App** (application), définit la forme géométrique de notre objet, tandis que la **partie graphique** (GUI), définit la façon dont l'objet sera affiché à l'écran.
L'outil **View Provider Object** (créateur de vue), comme toutes les fonctions FreeCAD, n'est disponible que lorsque vous exécutez FreeCAD dans son interface (GUI).

Il ya plusieurs manières et méthodes disponibles pour créer votre projet. Les méthodes utilisées doivent êtres une des méthodes prédéfinies que vous fourni FreeCAD, et apparaîtra dans la fenêtre **Propriété**, afin qu'ils puissent être modifiés par l'utilisateur (onglet **Données**).
De cette manière, les objets sont **FeaturePython** (ont toutes les

propriétés de Python) et sont totalements paramétriques. Vous pouvez paramétrer les **propriétés** et l'affichage **ViewObject** de l'objet séparément.

Astuce: dans les versions antérieures, nous avons utilisé le module Python cPickle (http://docs.python.org/release/2.5/lib/module-cPickle.html). Cependant, ce module exécute du code arbitrairement et provoque ainsi des problèmes de sécurité. Alors, nous avons opté pour le module Python json.

Exemples de base

L'exemple suivant (portion) peut être trouvé sur la page, src/Mod/TemplatePyMod/FeaturePython.py (http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/TemplatePyMod/FeaturePython.py?view=markup) qui inclus beaucoup d'autres exemples:

```
"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])

    def getIcon(self):
        '''Return the icon in XPM format which will appear in the tree view. This method is\'''
                '''optional and if not defined a default icon is shown.'''
        return """
            /* XPM */
            static const char * ViewProviderBox_xpm[] = {
            "16 16 6 1",
            "    c None",
            ".   c #141010",
            "+   c #615BD2",
            "@   c #C39D55",
            "#   c #000000",
            "$   c #57C355",
            "        ........",
            "   ......++..+..",
            "   .@@@@.++..++.",
            "   .@@@@.++..++.",
            "   .@@  .++++++.",
            "  ..@@  .++..++.",
            "###@@@@ .++..++.",
            "##$.@@$#.++++++.",
            "#$#$.$$$........",
            "#$$#######      ",
            "#$$#$$$$$#      ",
            "#$$#$$$$$#      ",
            "#$$#$$$$$#      ",
            " #$#$$$$$#      ",
            "  ##$$$$$#      ",
            "    #######     "};
            """
```

```
    def __getstate__(self):
        '''When saving the document this object gets stored using Python's json module.\'''
                '''Since we have some un-serializable parts here -- the Coin stuff -- we must define this
                '''to return a tuple of all serializable objects or None.'''"
        return None

    def __setstate__(self,state):
        '''When restoring the serialized object from document we have the chance to set some internals h
                '''Since no data were serialized nothing needs to be done here.'''"
        return None


def makeBox():
    FreeCAD.newDocument()
    a=FreeCAD.ActiveDocument.addObject("App::FeaturePython","Box")
    Box(a)
    ViewProviderBox(a.ViewObject)
```

Propriétés disponibles

Les **propriétés** sont **les bases** des **FeaturePython objets**.
Grâce à elles, l'utilisateur est en mesure d'interagir et de modifier son objet.
Après avoir créé un nouveau ObjetPython dans votre document **(a = FreeCAD.ActiveDocument.addObject ("App :: FeaturePython", "Box"))**, ses propriétés sont directement accessibles, vous pouvez obtenir la liste,
en faisant:

```
obj.supportedProperties()
```

Et voici, la liste des propriétés disponibles:

```
App::PropertyBool
App::PropertyBoolList
App::PropertyFloat
App::PropertyFloatList
App::PropertyFloatConstraint
App::PropertyQuantity
App::PropertyQuantityConstraint
App::PropertyAngle
App::PropertyDistance
App::PropertyLength
App::PropertySpeed
App::PropertyAcceleration
App::PropertyForce
App::PropertyPressure
App::PropertyInteger
App::PropertyIntegerConstraint
App::PropertyPercent
App::PropertyEnumeration
App::PropertyIntegerList
App::PropertyIntegerSet
App::PropertyMap
App::PropertyString
```

```
App::PropertyUUID
App::PropertyFont
App::PropertyStringList
App::PropertyLink
App::PropertyLinkSub
App::PropertyLinkList
App::PropertyLinkSubList
App::PropertyMatrix
App::PropertyVector
App::PropertyVectorList
App::PropertyPlacement
App::PropertyPlacementLink
App::PropertyColor
App::PropertyColorList
App::PropertyMaterial
App::PropertyPath
App::PropertyFile
App::PropertyFileIncluded
App::PropertyPythonObject
Part::PropertyPartShape
Part::PropertyGeometryList
Part::PropertyShapeHistory
Part::PropertyFilletEdges
Sketcher::PropertyConstraintList
```

Lors de l'ajout de propriétés à vos objets, prenez soin de ceci:

- Ne pas utiliser de caractères **"<"** ou **">"** dans les descriptions des propriétés (qui coupent des portions de code dans le fichier xml.Fcstd)
- Les propriétés sont stockées dans un fichier texte **.Fcstd**.
- Toutes les **propriétés** dont le nom vient après **"Shape"** sont triés dans l'ordre alphabétique, donc, si vous avez une forme dans vos propriétés, et comme les propriétés sont chargées après la forme, il peut y avoir des comportements inattendus!

==Property Type== By default the properties can be updated. It is possible to make the properties read-only, for instance in the case one wants to show the result of a method. It is also possible to hide the property. The property type can be set using

```
obj.setEditorMode("MyPropertyName", mode)
```

Mode est un **int court** qui peut avoir la valeur: 0 -- mode par défaut, lecture et écriture 1 -- lecture seule 2 -- caché

Autres exemples plus complexes

Cet exemple utilise le module Part Module pour créer un

octaèdre (http://fr.wikipedia.org/wiki/Octaèdre), puis crée sa représentation **coin (http://www.coin3d.org/) avec pivy**

En premier, c'est **l'objet document** lui-même:

```
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
         obj.addProperty("App::PropertyLength","Width","Octahedron","Width of the octahedron").Width=1.0
         obj.addProperty("App::PropertyLength","Height","Octahedron", "Height of the octahedron").Height=
         obj.addProperty("Part::PropertyPartShape","Shape","Octahedron", "Shape of the octahedron")
         obj.Proxy = self

      def execute(self, fp):
         # Define six vetices 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 mehod to create the faces
      def make_face(self,v1,v2,v3):
         wire = Part.makePolygon([v1,v2,v3,v1])
         face = Part.Face(wire)
         return face
```

Puis, nous avons **view provider object**, qui est responsable d'afficher l'objet dans la scène 3D (votre projet à l'écran):

```
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
         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)
            self.face.coordIndex.set1Value(10,2)
            self.face.coordIndex.set1Value(11,-1)

            self.face.coordIndex.set1Value(12,4)
            self.face.coordIndex.set1Value(13,0)
            self.face.coordIndex.set1Value(14,2)
            self.face.coordIndex.set1Value(15,-1)

            self.face.coordIndex.set1Value(16,1)
            self.face.coordIndex.set1Value(17,0)
            self.face.coordIndex.set1Value(18,5)
            self.face.coordIndex.set1Value(19,-1)

            self.face.coordIndex.set1Value(20,3)
            self.face.coordIndex.set1Value(21,1)
            self.face.coordIndex.set1Value(22,5)
            self.face.coordIndex.set1Value(23,-1)

            self.face.coordIndex.set1Value(24,4)
            self.face.coordIndex.set1Value(25,3)
            self.face.coordIndex.set1Value(26,5)
            self.face.coordIndex.set1Value(27,-1)

            self.face.coordIndex.set1Value(28,0)
            self.face.coordIndex.set1Value(29,4)
            self.face.coordIndex.set1Value(30,5)
            self.face.coordIndex.set1Value(31,-1)

    def getDisplayModes(self,obj):
        "Return a list of display modes."
        modes=[]
        modes.append("Shaded")
        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):
        return mode

    def onChanged(self, vp, prop):
        "Here we can do something when a single property got changed"
        FreeCAD.Console.PrintMessage("Change property: " + str(prop) + "\n")
        if prop == "Color":
            c = vp.getPropertyByName("Color")
            self.color.rgb.setValue(c[0],c[1],c[2])

    def getIcon(self):
        return """
            /* XPM */
            static const char * ViewProviderBox_xpm[] = {
            "16 16 6 1",
            "    c None",
            ".   c #141010",
            "+   c #615BD2",
            "@   c #C39D55",
            "#   c #000000",
            "$   c #57C355",
            "        ........",
            "   ......++..+..",
            "   .@@@@.++..++.",
            "   .@@@@.++..++.",
            "   .@@  .++++++.",
            "  ..@@  .++..++.",
            "###@@@@ .++..++.",
            "##$.@@$#.++++++.",
            "#$##$.$$$........",
            "#$$#######      ",
            "#$$#$$$$$#      ",
            "#$$#$$$$$#      ",
            "#$$#$$$$$#      ",
            " #$#$$$$$#      ",
            "  ##$$$$$#      ",
            "   #######      "};
            """

    def __getstate__(self):
        return None

    def __setstate__(self,state):
        return None
```

Enfin, une fois que notre objet et son **viewobject** sont définis, nous n'avons qu'a les appeler:

```
FreeCAD.newDocument()
    a=FreeCAD.ActiveDocument.addObject("App::FeaturePython","Octahedron")
    Octahedron(a)
    ViewProviderOctahedron(a.ViewObject)
```

Création d'objets sélectionnables

Si vous voulez travailler sur un objet sélectionné, ou du moins

une partie de celui-ci, vous cliquez sur l'objet dans la fenêtre, vous devez inclure la forme géométrique à l'intérieur d'un noeud **SoFCSelection node**.
Si votre objet a une représentation complexe, avec des widgets, des annotations, etc, vous pouvez n'inclure qu'une partie de celui-ci dans un **SoFCSelection**.
Tout ce qui est **SoFCSelection** est constamment "scanné" par FreeCAD pour voir s'il est sélectionné/présélectionné, il est donc logique de ne rien surcharger avec des **scans** inutiles.

Voici un exemple de ce que vous devrez faire pour inclure un **self.face**:

```
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
selectionNode.subElementName.setValue("Face")
selectNode.addChild(self.face)
...
self.shaded.addChild(selectionNode)
self.wireframe.addChild(selectionNode)
```

Vous créez Simplement un **SoFCSelection** node (noeud), puis vous lui ajoutez vos noeuds géométriques, alors seulement vous l'ajoutez à votre noeud principal, au lieu d'ajouter vos noeuds géométriques directement.

Travailler avec des formes simples

Si votre objet paramétrique renvoie simplement une forme, vous n'avez pas besoin d'utiliser un objet créateur de vue (*view provider object*).
La forme sera affichée à l'aide du module standard de représentation des formes de FreeCAD:

```
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
FreeCAD.ActiveDocument.recompute()
```

Same code with use **ViewProviderLine**

```
import FreeCAD as App
import FreeCADGui
import FreeCAD
import Part

class Line:
    def __init__(self, obj):
        '''"App two point properties" '''
        obj.addProperty("App::PropertyVector","p1","Line","Start point")
        obj.addProperty("App::PropertyVector","p2","Line","End point").p2=FreeCAD.Vector(100,0,0)
        obj.Proxy = self

    def execute(self, fp):
        '''"Print a short message when doing a recomputation, this method is mandatory" '''
        fp.Shape = Part.makeLine(fp.p1,fp.p2)

class ViewProviderLine:
   def __init__(self, obj):
      ''' Set this object to the proxy object of the actual view provider '''
      obj.Proxy = self

   def getDefaultDisplayMode(self):
      ''' Return the name of the default display mode. It must be defined in getDisplayModes. '''
      return "Flat Lines"

a=FreeCAD.ActiveDocument.addObject("Part::FeaturePython","Line")
Line(a)
ViewProviderLine(a.ViewObject)
App.ActiveDocument.recompute()
```

< précédent: PySide Index suivant: Embedding FreeCAD >

FreeCAD a la capacité incroyable de pouvoir être importé **en tant que module Python** dans d'autres programmes ou, dans une console Python autonome, avec tous ses modules et ses composants. Il est même possible d'importer l'interface graphique (GUI) de FreeCAD en tant que module python avec toutefois, **quelques restrictions**.

Utilisation de FreeCAD sans interface graphique (GUI)

Une première application, directe, facile et utile que vous pouvez faire est d'importer des documents FreeCAD dans votre programme. Dans l'exemple suivant, nous allons importer **Part geometry** d'un document FreeCAD dans blender (http://www.blender.org). Voici le script complet.

J'espère que vous serez impressionné par sa simplicité:

```
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%t|FreeCAD library not found. Please check the FREECADPATH variable in
   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()
```

Première chose, s'assurer que Python va trouver notre bibliothèque FreeCAD. Une fois qu'il l'a trouvée, tous les modules FreeCAD comme **Part**, que nous allons aussi utiliser, seront

disponibles automatiquement.

Donc, nous utilisons tout simplement la variable **sys.path**, qui va donner à Python le chemin des modules à rechercher, et nous ajoutons le chemin **FreeCAD lib**. Cette modification n'est que temporaire, et sera perdue quand nous aurons terminé avec notre interpréteur Python. Une autre façon, est de créer un lien vers votre bibliothèque FreeCAD dans l'un des chemins (Path) de recherche Python. Nous placerons le chemin dans une constante (**FREECADPATH**), un autre utilisateur du script aura ainsi plus de facilité pour configurer son propre système.

Une fois certain que la bibliothèque a été chargée (the try/except sequence), nous pourrons travailler avec FreeCAD, de la même manière que si nous le ferions à l'intérieur de l’interpréteur Python de FreeCAD. Nous ouvrons le document FreeCAD que nous avons chargé avec la fonction **main()**, et nous listons ses objets. Puis, comme nous avons choisi de nous occuper que de la forme géométrique, nous vérifions si la propriété **Type** de chaque objet contient **Part**, puis nous faison une tesselation (http://fr.wikipedia.org/wiki/Tesselation).

La tesselation produit une liste de **sommets** (Vertex) et une liste de **faces** définis par les indices de sommets. C'est parfait, puisque c'est exactement de cette manière que Blender définit les mailles. Donc, notre tâche est ridiculement simple, nous ajoutons juste les deux listes des **sommets** et **faces** comme un maillage de Blender. Une fois fait, nous allons juste redessiner l'écran et, c'est fini !

Vous avez vu, ce script est très simple (en fait, j'en ai écris un plus évolué ici (http://yorik.orgfree.com/scripts /import_freecad.py)), vous voudrez peut-être l'étendre, par exemple importer des objets "mesh", ou importer "Part geometry" qui n'a pas de face, ou importer d'autres formats que FreeCAD peut lire. Vous pouvez également exporter les formes géométriques dans un document FreeCAD, la procédure est la même. Vous pouvez également créer un dialogue, afin que l'utilisateur puisse choisir ce qu'il veut importer, etc . . . En réalité, la beauté dans tout cela, réside du fait que vous laissez

faire la totalité du travail à FreeCAD, tout en présentant ses résultats dans le programme de votre choix.

Utilisation de FreeCAD avec interface graphique (GUI)

Depuis la version 4.2 de Qt, Qt a la capacité d'intégrer des plugins **Qt-GUI** dépendants d'applications hôtes non-Qt, et, de partager la boucle évènementielle de l'hôte.

Principalement pour FreeCAD, cela signifie qu'il peut être importé à partir d'une autre application avec son interface utilisateur entière (GUI) par conséquences, l'application hôte prend le contrôle total de FreeCAD.

L'ensemble du code Python nécessaire pour atteindre ce but, n'a que deux lignes:

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

Si, l'application hôte est basée sur Qt, alors cette solution devrait fonctionner sur toutes les plates-formes supportées par Qt. **Toutefois**, l'hôte doit être de la même version Qt que la version utilisée pour FreeCAD, sinon, vous pouvez obtenir des erreurs d'exécution inattendues.

Cependant, pour les applications non-Qt, il ya quelques restrictions, que vous devez connaitre:

- Cette solution ne fonctionnera probablement pas avec tous les autres outils (toolkit):
  - Pour Windows, il fonctionnera aussi longtemps que l'application hôte utilisée est compatible avec **Win32** ou, tout autres outils (toolkit) qui utilisent l'**API Win32**, comme **wxWidgets**, **MFC** ou **WinForms**.
  - Pour le faire fonctionner sous **X11 (http://fr.wikipedia.org/wiki/X_Window_System)** (Linux), l'application hôte doit utiliser la bibliothèque **"glib (http://developer.gnome.org/glib/)"**.

PS:pour toute application console, cette solution, bien sûr ne fonctionnera pas car, il n'y a pas de fonctionnement "boucle évènementielle" dans ce système.

< précédent: Scripted objects Index suivant: Code snippets >

Cette page contient, des exemples, des extraits de code en Python FreeCAD, recueillis auprès d'utilisateurs expérimentés et de produits de discussions sur les forums (http://forum.freecadweb.org/).

Lisez les et utilisez les comme point de départ pour vos propres scripts . .

Un fichier typique InitGui.py

En plus de votre module principal, chaque module doit contenir, un fichier **InitGui.py**, responsable de l'insertion du module dans l'interface principale.

Ceci est un simple exemple.

```
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())
```

Un fichier module typique

Ceci est l'exemple d'un fichier module principal, il contient tout ce que fait votre module. C'est le fichier **Scripts.py** invoqué dans l'exemple précédent. Vous avez ici toutes vos commandes personnalisées.

```
import FreeCAD, FreeCADGui

class ScriptCmd:
   def Activated(self):
       # Here your 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 detai

FreeCADGui.addCommand('Script_Cmd', ScriptCmd())
```

Importer un nouveau type de fichier

Importer un nouveau type de fichier dans FreeCAD est facile. FreeCAD ne prends pas en considération l'importation de n'importe quelle données dans un document ouvert, parce que, vous ne pouvez pas ouvrir directement un nouveau type de fichier.

Donc, ce que vous devez faire, c'est ajouter la nouvelle extension de fichier à la liste des extensions connues de FreeCAD, et, d'écrire le code qui va lire le fichier et créer les objets FreeCAD que vous voulez.

Cette ligne doit être ajoutée au fichier **InitGui.py** pour ajouter la nouvelle extension de fichier à la liste:

```
# 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")
```

Puis, dans le fichier **Import_Ext.py**, faites:

```
def open(filename):
   doc=App.newDocument()
   # here you do all what is needed with filename, read, classify data, create corresponding FreeCAD obje
   doc.recompute()
```

Pour **exporter** votre document avec une nouvelle extension, le fonctionnement est le même, mais vous devrez faire:

```
FreeCAD.addExportType("Your new File Type (*.ext)","Export_Ext")
```

Ajouter une ligne

Une ligne, à uniquement deux 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()
```

Ajouter un polygone

Un polygone est simplement un ensemble de segments connnectés (un polyline dans AutoCAD) il n'est pas obligatoirement fermé.

```
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()
```

Ajout et suppression d'objet(s) dans un groupe

```
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
```

PS: vous pouvez aussi ajouter un groupe dans un groupe . . .

Ajout d'une maille (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 langth of 100
m.scale(100.0)
# add the mesh to the active document
me=doc.addObject("Mesh::Feature","Cube")
me.Mesh=m
```

Ajout d'un arc ou d'un cercle

```
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()
```

Accéder et changer la représentation d'un objet

Chaque objet dans un document FreeCAD a un objet **vue** associé a une **représentation** qui stocke tous les paramètres qui définissent les propriétés de l'objet, comme, la couleur, l'épaisseur de la ligne, 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
```

Observation des évènements de la souris dans la vue 3D via Python

Le cadre **Inventor** permet d'ajouter un ou plusieurs noeuds (nodes) de rappel à la scène graphique visualisée. Par défaut, FreeCAD, possède un noeud (node) de rappel installé par la visionneuse (fenêtre d'affichage des graphes), qui permet d'ajouter des fonctions statiques ou globales en C++. Des méthodes de liaisons appropriées sont fournies avec Python, pour permettre l'utilisation de cette technique à partir de codes Python.

```
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)
```

Maintenant, choisissez une zone dans l'écran (surface de travail) 3D et observez les messages affichés dans la fenêtre de sortie.

Pour terminer l'observation il suffit de faire:

```
v.removeEventCallback("SoMouseButtonEvent",c)
```

Les types d'évènements suivants sont pris en charge:

- **SoEvent** -- tous types d'évènements
- **SoButtonEvent** -- tous les évènements, boutons, molette
- **SoLocation2Event** -- tous les évènements 2D (déplacements normaux de la souris)
- **SoMotion3Event** -- tous les évènements 3D (pour le spaceball)
- **SoKeyboardEvent** -- évènements des touches flèche haut et flèche bas
- **SoMouseButtonEvent** -- tous les évènements boutons Haut et Bas de la souris
- **SoSpaceballButtonEvent** -- tous les évènements Haut et Bas (pour le spaceball)

Les fonctions Python qui peuvent être enregistrées avec **addEventCallback()** attendent la définition d'une bibliothèque.

Suivant la façon dont l'évènement survient, la bibliothèque peut disposer de différentes clefs.

Il y a une clef pour chaque événement:

- **Type** -- le nom du type d'évènement par exemple **SoMouseEvent, SoLocation2Event, ...**
- **Time** -- l'heure courante codée dans une chaîne **string**
- **Position** -- un tuple de deux **integers (http://docs.python.org/library/functions.html#int)**, donant la position x,y de la souris
- **ShiftDown** -- type boolean, **true** si Shift est pressé sinon, **false**
- **CtrlDown** -- type boolean, **true** si Ctrl est pressé sinon, **false**

- **AltDown** -- type boolean, **true** si Alt est pressé sinon, **false**

Pour un évènement bouton comme clavier, souris ou spaceball

- **State** -- la chaîne **UP** si le bouton est relevé, **DOWN** si le bouton est enfoncé ou **UNKNOWN** si rien ne se passe

Pour un évènement clavier:

- **Key** -- le caractère de la touche qui est pressée

Pour un évènement bouton de souris:

- **Button** -- le bouton pressé peut être BUTTON1, ..., BUTTON5 ou tous

Pour un évènement spaceball:

- **Button** -- le bouton pressé peut être BUTTON1, ..., BUTTON7 ou tous

Et finalement les évènement de mouvements:

- **Translation** -- un tuple de trois **float() (http://docs.python.org/library/functions.html#float)**
- **Rotation** -- un quaternion, tuple de quattre **float() (http://docs.python.org/library/functions.html#float)**

Manipulation de scènes graphiques en Python

Il est aussi possible d'afficher ou de changer de scène en programmation Python, avec le module **pivy** en combinaison avec Coin (http://www.coin3d.org/)

```
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()
```

L'API Python de pivy est créé en utilisant l'outil SWIG (http://www.swig.org/). Comme dans FreeCAD nous utilisons

certains noeuds (nodes) écrits automatiquement nous ne pouvons pas les créer directement en Python. Il est cependant, possible de créer un noeud avec son nom interne. Un exemple de **SoFCSelection**, le **type** peut être créé avec:

```
type = SoType.fromName("SoFCSelection")
node = type.createInstance()
```

Ajouter et effacer des objets de la scène

Ajouter de nouveaux noeuds dans la scène graphique peut être fait de cette façon. Prenez toujours soin d'ajouter un **SoSeparator** pour, contenir les propriétés de la forme géométrique, les coordonnées et le matériel d'un même objet. L'exemple suivant ajoute une ligne rouge à partir de (0,0,0) à (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)
```

Pour le supprimer, il suffit de:

```
sg.removeChild(no)
```

Ajout de widgets personnalisés à l'interface

Vous pouvez créer un widget avec Qt designer (http://fr.wikipedia.org/wiki/Qt), le transformer en Script Python et l'incorporer dans l'interface de FreeCAD avec PySide.

Généralement codé comme ceci (il est simple, vous pouvez aussi le coder directement en 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()))

   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.QApplicatio
   self.label.setText(QtGui.QApplication.translate("myWidget", "Welcome to my new widget!", None, QtGui.Q
```

Puis, vous devez créer une référence à la fenêtre FreeCAD Qt, lui insérer le widget personnalisé, et transférer le code Ui du widget que nous venons de faire dans le vôtre avec:

```
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
```

- Ici, le code Python est généré par le compilateur **Ui Python** avec le module **pyuic.py** (il existe aussi pyuic4.py attention à la compatibilité).
  - Vous pouvez trouver ce fichier à l'emplacement **"C:\Program Files\FreeCAD0.13\bin\PyQt4\uic"**,
- **pyuic.py** est l'outil qui convertit les fichiers qt-designer **.ui** (Interface Utilisateur) en fichier **.py** (code Python), la ligne de commande dans la console DOS est **"pyuic -x fichier.ui > fichier.py"**
- vous pouvez créer un fichier .bat pour automatiser la commande:
- (avec Python27) copier cette ligne dans un fichier texte, et, le sauver le sous le nom **"compile.bat"**

```
@"C:\Python27\python" "C:\Python27\Lib\site-packages\PyQt4\uic\pyuic.py" -x %1.ui > %1.py
```

(au besoin, adaptez le chemin à votre version de Python)

Si vous utilisez les outils fourni dans FreeCAD, le code sera,

```
@"C:\Program Files\FreeCAD0.13\bin\python" "C:\Program Files\FreeCAD0.13\bin\PyQt4\uic\pyuic.py" -x %1.u
```

- et tapez à la ligne de commande " **compile fichier** " sans extension, le nom "**fichier**" entré **.ui**, sera le nom sortant avec extension **.py**
- **ATTENTION: il faut que les fichiers soient présents, et, accessibles, vérifiez que les fichiers sont présents et que les chemins sont justes !**
- pour cet exemple entièrement automatique et simplifié, **"compile.bat"** est au même endroit que le **fichier.ui** à convertir en **fichier.py**

Autres liens de documentation "Python and Qt" (http://www.qtrac.eu/pyqtbook.html) , sur Développez.com (http://ogirardot.developpez.com/introduction-pyqt/) et bien d'autres.

Vous pouvez installer une version complète de Python qui comprend **PyQt, Qt Designer ...** (http://www.riverbankcomputing.co.uk/software/pyqt/download)

Ajout d'une liste déroulante

Le code suivant vous permet d'ajouter une liste déroulante dans FreeCAD, en plus des onglets "Projet" et "tâches".

Il utilise également le module **uic** pour charger un fichier **ui** directement dans cet onglet.

```
# create new Tab in ComboView
from PySide import QtGui,QtCore
#from PySide import uic

def getMainWindow():
   "returns the main window"
   # using QtGui.qApp.activeWindow() isn't very reliable because if another
   # widget than the mainwindow is active (e.g. a dialog) the wrong widget is
   # returned
   toplevel = QtGui.qApp.topLevelWidgets()
   for i in toplevel:
       if i.metaObject().className() == "Gui::MainWindow":
           return i
   raise Exception("No main window found")

def getComboView(mw):
   dw=mw.findChildren(QtGui.QDockWidget)
   for i in dw:
       if str(i.objectName()) == "Combo View":
           return i.findChild(QtGui.QTabWidget)
       elif str(i.objectName()) == "Python Console":
```

```
            return i.findChild(QtGui.QTabWidget)
    raise Exception ("No tab widget found")

mw = getMainWindow()
tab = getComboView(getMainWindow())
tab2=QtGui.QDialog()
tab.addTab(tab2,"A Special Tab")

#uic.loadUi("/myTaskPanelforTabs.ui",tab2)
tab2.show()
#tab.removeTab(2)
```

Enable or disable a window

```
from PySide import QtGui
mw=FreeCADGui.getMainWindow()
dws=mw.findChildren(QtGui.QDockWidget)

# objectName may be :
# "Report view"
# "Tree view"
# "Property view"
# "Selection view"
# "Combo View"
# "Python console"
# "draftToolbar"

for i in dws:
  if i.objectName() == "Report view":
    dw=i
    break

va=dw.toggleViewAction()
va.setChecked(True)        # True or False
dw.setVisible(True)        # True or False
```

Ouverture d'une page web

```
import WebGui
WebGui.openBrowser("http://www.example.com")
```

Obtenir le code HTML d'une page Web ouverte

```
from PyQt4 import QtGui,QtWebKit
a = QtGui.qApp
mw = a.activeWindow()
v = mw.findChild(QtWebKit.QWebFrame)
html = unicode(v.toHtml())
print html
```

Extraire et utiliser les coordonnées de 3 points sélectionnés

```
# -*- coding: utf-8 -*-
# the line above to put the accentuated in the remarks
# If this line is missing, an error will be returned
# extract and use the coordinates of 3 objects selected
import Part, FreeCAD, math, PartGui, FreeCADGui
from FreeCAD import Base, Console
sel = FreeCADGui.Selection.getSelection() # " sel " contains the items selected
if len(sel)!=3 :
  # If there are no 3 objects selected, an error is displayed in the report view
  # The \r and \n at the end of line mean return and the newline CR + LF.
  Console.PrintError("Select 3 points exactly\r\n")
else :
  points=[]
  for obj in sel:
    points.append(obj.Shape.BoundBox.Center)

  for pt in points:
    # display of the coordinates in the report view
    Console.PrintMessage(str(pt.x)+"\r\n")
    Console.PrintMessage(str(pt.y)+"\r\n")
    Console.PrintMessage(str(pt.z)+"\r\n")

  Console.PrintMessage(str(pt[1]) + "\r\n")
```

Lister les objets

```
# -*- coding: utf-8 -*-
import FreeCAD,Draft
# List all objects of the document
doc = FreeCAD.ActiveDocument
objs = FreeCAD.ActiveDocument.Objects
#App.Console.PrintMessage(str(objs) + "\n")
#App.Console.PrintMessage(str(len(FreeCAD.ActiveDocument.Objects)) + " Objects"  + "\n")

for obj in objs:
    a = obj.Name                                        # list the Name  of the object  (not modifia
    b = obj.Label                                       # list the Label of the object  (modifiable,
    try:
        c = obj.LabelText                               # list the LabeText of the text (modifiable,
        App.Console.PrintMessage(str(a) +" "+ str(b) +" "+ str(c) + "\n") # Displays the Name the Label a
    except:
        App.Console.PrintMessage(str(a) +" "+ str(b) + "\n") # Displays the Name and the Label of the ob

#doc.removeObject("Box") # Clears the designated object
```

Fonction résidente avec action au clic de souris

```
# -*- coding: utf-8 -*-
# causes an action to the mouse click on an object
# This function remains resident (in memory) with the function "addObserver(s)"
# "removeObserver(s) # Uninstalls the resident function
class SelObserver:
    def addSelection(self,doc,obj,sub,pnt):               # Selection object
    #def setPreselection(self,doc,obj,sub):                # Preselection object
        App.Console.PrintMessage("addSelection"+ "\n")
        App.Console.PrintMessage(str(doc)+ "\n")          # Name of the document
        App.Console.PrintMessage(str(obj)+ "\n")          # Name of the object
        App.Console.PrintMessage(str(sub)+ "\n")          # The part of the object name
        App.Console.PrintMessage(str(pnt)+ "\n")          # Coordinates of the object
        App.Console.PrintMessage("______"+ "\n")

    def removeSelection(self,doc,obj,sub):                # Delete the selected object
```

```
        App.Console.PrintMessage("removeSelection"+ "\n")
    def setSelection(self,doc):                         # Selection in ComboView
        App.Console.PrintMessage("setSelection"+ "\n")
    def clearSelection(self,doc):                       # If click on the screen, clear the selection
        App.Console.PrintMessage("clearSelection"+ "\n")  # If click on another object, clear the previo
s =SelObserver()
FreeCADGui.Selection.addObserver(s)                     # install the function mode resident
#FreeCADGui.Selection.removeObserver(s)                 # Uninstall the resident function
```

Lister les composantes d'un objet

```
# -*- coding: utf-8 -*-
# This function list the components of an object
# and extract this object its XYZ coordinates,
# its edges and their lengths center of mass and coordinates
# its faces and their center of mass
# its faces and their surfaces and coordinates
# 8/05/2014

import Draft,Part
def detail():
    sel = FreeCADGui.Selection.getSelection()   # Select an object
    if len(sel) != 0:                           # If there is a selection then
        Vertx=[]
        Edges=[]
        Faces=[]
        compt_V=0
        compt_E=0
        compt_F=0
        pas    =0
        perimetre = 0.0
        EdgesLong = []

        # Displays the "Name" and the "Label" of the selection
        App.Console.PrintMessage("Selection > " + str(sel[0].Name) + "  " + str(sel[0].Label) +"\n"+"\n"

        for j in enumerate(sel[0].Shape.Edges):                                     # Search the "Edges"
            compt_E+=1
            Edges.append("Edge%d" % (j[0]+1))
            EdgesLong.append(str(sel[0].Shape.Edges[compt_E-1].Length))
            perimetre += (sel[0].Shape.Edges[compt_E-1].Length)                     # calculates the per

            # Displays the "Edge" and its length
            App.Console.PrintMessage("Edge"+str(compt_E)+" Length > "+str(sel[0].Shape.Edges[compt_E-1].L

            # Displays the "Edge" and its center mass
            App.Console.PrintMessage("Edge"+str(compt_E)+" Center > "+str(sel[0].Shape.Edges[compt_E-1].C

            num = sel[0].Shape.Edges[compt_E-1].Vertexes[0]
            Vertx.append("X1: "+str(num.Point.x))
            Vertx.append("Y1: "+str(num.Point.y))
            Vertx.append("Z1: "+str(num.Point.z))
            # Displays the coordinates 1
            App.Console.PrintMessage("X1: "+str(num.Point[0])+" Y1: "+str(num.Point[1])+" Z1: "+str(num.P

            try:
                num = sel[0].Shape.Edges[compt_E-1].Vertexes[1]
                Vertx.append("X2: "+str(num.Point.x))
                Vertx.append("Y2: "+str(num.Point.y))
                Vertx.append("Z2: "+str(num.Point.z))
            except:
                Vertx.append("-")
                Vertx.append("-")
                Vertx.append("-")
            # Displays the coordinates 2
            App.Console.PrintMessage("X2: "+str(num.Point[0])+" Y2: "+str(num.Point[1])+" Z2: "+str(num.P
```

```
            App.Console.PrintMessage("\n")
        App.Console.PrintMessage("Perimeter of the form  : "+str(perimetre)+"\n")

        App.Console.PrintMessage("\n")
        FacesSurf = []
        for j in enumerate(sel[0].Shape.Faces):                                    # Search the "Faces"
            compt_F+=1
            Faces.append("Face%d" % (j[0]+1))
            FacesSurf.append(str(sel[0].Shape.Faces[compt_F-1].Area))

            # Displays 'Face' and its surface
            App.Console.PrintMessage("Face"+str(compt_F)+" >  Surface "+str(sel[0].Shape.Faces[compt_F-1]

            # Displays 'Face' and its CenterOfMass
            App.Console.PrintMessage("Face"+str(compt_F)+" >  Center  "+str(sel[0].Shape.Faces[compt_F-1]

            # Displays 'Face' and its Coordinates
            FacesCoor = []
            fco = 0
            for f0 in sel[0].Shape.Faces[compt_F-1].Vertexes:                      # Search the Vertex
                fco += 1
                FacesCoor.append("X"+str(fco)+": "+str(f0.Point.x))
                FacesCoor.append("Y"+str(fco)+": "+str(f0.Point.y))
                FacesCoor.append("Z"+str(fco)+": "+str(f0.Point.z))

            # Displays 'Face' and its Coordinates
            App.Console.PrintMessage("Face"+str(compt_F)+" >  Coordinate"+str(FacesCoor)+"\n")

            # Displays 'Face' and its Volume
            App.Console.PrintMessage("Face"+str(compt_F)+" >  Volume  "+str(sel[0].Shape.Faces[compt_F-1]
            App.Console.PrintMessage("\n")

        # Displays the total surface of the form
        App.Console.PrintMessage("Surface of the form    : "+str(sel[0].Shape.Area)+"\n")

        # Displays the total Volume of the form
        App.Console.PrintMessage("Volume  of the form    : "+str(sel[0].Shape.Volume)+"\n")

detail()
```

Lister les PropertiesList

```
import FreeCADGui
from FreeCAD import Console
o = App.ActiveDocument.ActiveObject
op = o.PropertiesList
for p in op:
    Console.PrintMessage("Property: "+ str(p)+ " Value: " + str(o.getPropertyByName(p))+"\r\n")
```

Search and data extraction

Examples of research and decoding information on an object.

Each section is independently and is separated by "############" can be copied directly into the Python console, or in a macro or use this macro. The description of the macro in the commentary.

Displaying it in the "View Report" window (View > Views > View report)

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

# Exemples de recherche et de decodage d'informations sur un objet
# Chaque section peut etre copiee directement dans la console Python ou dans une macro ou utilisez la mac
# certaines commandes se repetent seul l'approche est differente
#
# Examples of research and decoding information on an object
# Each section can be copied directly into the Python console, or in a macro or uses this macro
# Certain commands as repeat alone approach is different
#
# rev:29/09/2014

from FreeCAD import Base
import DraftVecUtils, Draft, Part

mydoc = FreeCAD.activeDocument().Name                                   # Name of active Document
App.Console.PrintMessage("Active docu    : "+str(mydoc)+"\n")
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelect
object_Label = sel[0].Label                                             # Label of the object (modifial
App.Console.PrintMessage("object_Label   : "+str(object_Label)+"\n")
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelect
App.Console.PrintMessage("sel            : "+str(sel[0])+"\n\n")          # sel[0] first object selecte
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelect
object_Name  = sel[0].Name                                              # Name of the object (not modi
App.Console.PrintMessage("object_Name    : "+str(object_Name)+"\n\n")
###############################################################################

try:
    SubElement = FreeCADGui.Selection.getSelectionEx()                  # sub element name with getSel
    element_ = SubElement[0].SubElementNames[0]                         # name of 1 element selected
    App.Console.PrintMessage("elementSelec   : "+str(element_)+"\n\n")
except:
    App.Console.PrintMessage("Oups"+"\n\n")
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelect
App.Console.PrintMessage("sel            : "+str(sel[0])+"\n\n")        # sel[0] first object selected
###############################################################################

SubElement = FreeCADGui.Selection.getSelectionEx()                      # sub element name with getSel
App.Console.PrintMessage("SubElement     : "+str(SubElement[0])+"\n\n")     # name of sub element
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelect
i = 0
for j in enumerate(sel[0].Shape.Edges):                                 # list all Edges
    i += 1
    App.Console.PrintMessage("Edges n : "+str(i)+"\n")
    a = sel[0].Shape.Edges[j[0]].Vertexes[0]
    App.Console.PrintMessage("X1             : "+str(a.Point.x)+"\n")     # coordinate XYZ first point
    App.Console.PrintMessage("Y1             : "+str(a.Point.y)+"\n")
    App.Console.PrintMessage("Z1             : "+str(a.Point.z)+"\n")
    try:
        a = sel[0].Shape.Edges[j[0]].Vertexes[1]
        App.Console.PrintMessage("X2             : "+str(a.Point.x)+"\n") # coordinate XYZ second point
        App.Console.PrintMessage("Y2             : "+str(a.Point.y)+"\n")
        App.Console.PrintMessage("Z2             : "+str(a.Point.z)+"\n")
    except:
        App.Console.PrintMessage("Oups"+"\n")
```

```
App.Console.PrintMessage("\n")
################################################################################

try:
    SubElement = FreeCADGui.Selection.getSelectionEx()                    # sub element name with getSele
    subElementName = Gui.Selection.getSelectionEx()[0].SubElementNames[0] # sub element name with getSele
    App.Console.PrintMessage("subElementName : "+str(subElementName)+"\n")

    subObjectX = Gui.Selection.getSelectionEx()[0].SubObjects[0].Point.x  # sub element coordinate X
    App.Console.PrintMessage("subObject_X    : "+str(subObjectX)+"\n")
    subObjectY = Gui.Selection.getSelectionEx()[0].SubObjects[0].Point.y  # sub element coordinate Y
    App.Console.PrintMessage("subObject_Y    : "+str(subObjectY)+"\n")
    subObjectZ = Gui.Selection.getSelectionEx()[0].SubObjects[0].Point.z  # sub element coordinate Z
    App.Console.PrintMessage("subObject_Z    : "+str(subObjectZ)+"\n")

    subObjectLength = Gui.Selection.getSelectionEx()[0].SubObjects[0].Length # sub element Length
    App.Console.PrintMessage("subObjectLength: "+str(subObjectLength)+"\n")

    surfaceFace = Gui.Selection.getSelectionEx()[0].SubObjects[0].Area    # Area of the 1 face
    App.Console.PrintMessage("surfaceFace    : "+str(surfaceFace)+"\n\n")
except:
    App.Console.PrintMessage("Oups"+"\n\n")
################################################################################

sel = FreeCADGui.Selection.getSelection()                                 # select object with getSelect
surface = sel[0].Shape.Area                                               # Area object complete
App.Console.PrintMessage("surfaceObjet   : "+str(surface)+"\n\n")
################################################################################

sel = FreeCADGui.Selection.getSelection()                                 # select object with getSelect
CenterOfMass = sel[0].Shape.CenterOfMass                                  # Center of Mass of the object
App.Console.PrintMessage("CenterOfMass   : "+str(CenterOfMass)+"\n")
App.Console.PrintMessage("CenterOfMassX  : "+str(CenterOfMass[0])+"\n")   # coordinates [0]=X [1]=Y [2]=
App.Console.PrintMessage("CenterOfMassY  : "+str(CenterOfMass[1])+"\n")
App.Console.PrintMessage("CenterOfMassZ  : "+str(CenterOfMass[2])+"\n\n")
################################################################################

sel = FreeCADGui.Selection.getSelection()                                 # select object with getSelect
for j in enumerate(sel[0].Shape.Faces):                                   # List alles faces of the obje
    App.Console.PrintMessage("Face           : "+str("Face%d" % (j[0]+1))+"\n")
App.Console.PrintMessage("\n\n")
################################################################################

sel = FreeCADGui.Selection.getSelection()                                 # select object with getSelect
volume_ = sel[0].Shape.Volume                                             # Volume of the object
App.Console.PrintMessage("volume_        : "+str(volume_)+"\n\n")
################################################################################

sel = FreeCADGui.Selection.getSelection()                                 # select object with getSelect
boundBox_= sel[0].Shape.BoundBox                                          # BoundBox of the object
App.Console.PrintMessage("boundBox_      : "+str(boundBox_)+"\n")

boundBoxLX = boundBox_.XLength                                            # Length x boundBox rectangle
boundBoxLY = boundBox_.YLength                                            # Length y boundBox rectangle
boundBoxLZ = boundBox_.ZLength                                            # Length z boundBox rectangle
App.Console.PrintMessage("boundBoxLX     : "+str(boundBoxLX)+"\n")
App.Console.PrintMessage("boundBoxLY     : "+str(boundBoxLY)+"\n")
App.Console.PrintMessage("boundBoxLZ     : "+str(boundBoxLZ)+"\n\n")
################################################################################

sel = FreeCADGui.Selection.getSelection()                                 # select object with getSelect
pl = sel[0].Shape.Placement                                               # Placement Vector XYZ and Yaw
App.Console.PrintMessage("Placement      : "+str(pl)+"\n")
################################################################################

sel = FreeCADGui.Selection.getSelection()                                 # select object with getSelect
pl = sel[0].Shape.Placement.Base                                          # Placement Vector XYZ
App.Console.PrintMessage("PlacementBase  : "+str(pl)+"\n\n")
################################################################################

sel = FreeCADGui.Selection.getSelection()                                 # select object with getSelect
Yaw = sel[0].Shape.Placement.Rotation.toEuler()[0]                        # decode angle Euler Yaw
```

```
App.Console.PrintMessage("Yaw            : "+str(Yaw)+"\n")
Pitch = sel[0].Shape.Placement.Rotation.toEuler()[1]                    # decode angle Euler Pitch
App.Console.PrintMessage("Pitch          : "+str(Pitch)+"\n")
Roll = sel[0].Shape.Placement.Rotation.toEuler()[2]                     # decode angle Euler Yaw
App.Console.PrintMessage("Yaw            : "+str(Roll)+"\n\n")
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelecti
oripl_X = sel[0].Placement.Base[0]                                      # decode Placement X
oripl_Y = sel[0].Placement.Base[1]                                      # decode Placement Y
oripl_Z = sel[0].Placement.Base[2]                                      # decode Placement Z

App.Console.PrintMessage("oripl_X        : "+str(oripl_X)+"\n")
App.Console.PrintMessage("oripl_Y        : "+str(oripl_Y)+"\n")
App.Console.PrintMessage("oripl_Z        : "+str(oripl_Z)+"\n\n")
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelecti
rotation = sel[0].Placement.Rotation                                    # decode Placement Rotation
App.Console.PrintMessage("rotation             : "+str(rotation)+"\n\n")
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelecti
pl = sel[0].Shape.Placement.Rotation                                    # decode Placement Rotation otl
App.Console.PrintMessage("Placement Rot        : "+str(pl)+"\n\n")
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelecti
pl = sel[0].Shape.Placement.Rotation.Angle                              # decode Placement Rotation Ang
App.Console.PrintMessage("Placement Rot Angle   : "+str(pl)+"\n\n")
###############################################################################

sel = FreeCADGui.Selection.getSelection()                               # select object with getSelecti
Rot_0 = sel[0].Placement.Rotation.Q[0]                                  # decode Placement Rotation 0
App.Console.PrintMessage("Rot_0         : "+str(Rot_0)+ " rad ,  "+str(180 * Rot_0 / 3.1416)+" deg "+"\n

Rot_1 = sel[0].Placement.Rotation.Q[1]                                  # decode Placement Rotation 1
App.Console.PrintMessage("Rot_1         : "+str(Rot_1)+ " rad ,  "+str(180 * Rot_1 / 3.1416)+" deg "+"\n

Rot_2 = sel[0].Placement.Rotation.Q[2]                                  # decode Placement Rotation 2
App.Console.PrintMessage("Rot_2         : "+str(Rot_2)+ " rad ,  "+str(180 * Rot_2 / 3.1416)+" deg "+"\n

Rot_3 = sel[0].Placement.Rotation.Q[3]                                  # decode Placement Rotation 3
App.Console.PrintMessage("Rot_3         : "+str(Rot_3)+"\n\n")
###############################################################################
```

Manual search of an element with label

```
# Extract the coordinate X,Y,Z and Angle giving the label
App.Console.PrintMessage("Base.x       : "+str(FreeCAD.ActiveDocument.getObjectsByLabel("Cylindre")[0].P
App.Console.PrintMessage("Base.y       : "+str(FreeCAD.ActiveDocument.getObjectsByLabel("Cylindre")[0].P
App.Console.PrintMessage("Base.z       : "+str(FreeCAD.ActiveDocument.getObjectsByLabel("Cylindre")[0].P
App.Console.PrintMessage("Base.Angle   : "+str(FreeCAD.ActiveDocument.getObjectsByLabel("Cylindre")[0].P
###############################################################################
```

PS: Usually the angles are given in Radian to convert :

1. angle in Degrees to Radians :
   - Angle in radian = **pi * (angle in degree) / 180**
   - Angle in radian = math.radians(angle in degree)
2. angle in Radians to Degrees :

- Angle in degree = **180 * (angle in radian) / pi**
- Angle in degree = math.degrees(angle in radian)

Cartesian coordinates

This code displays the Cartesian coordinates of the selected item.

Change the value of "numberOfPoints" if you want a different number of points (precision)

```
numberOfPoints = 100                                                          # Decomposition number (or p
selectedEdge = FreeCADGui.Selection.getSelectionEx()[0].SubObjects[0].copy() # select one element
points  = selectedEdge.discretize(numberOfPoints)                             # discretize the element
i=0
for p in points:                                                              # list and display the coord
    i+=1
    print i, " X", p.x, " Y", p.y, " Z", p.z
```

Other method display on "Int" and "Float"

```
import Part
from FreeCAD import Base

c=Part.makeCylinder(2,10)        # create the circle
Part.show(c)                     # display the shape

# slice accepts two arguments:
#+ the normal of the cross section plane
#+ the distance from the origin to the cross section plane. Here you have to find a value so that the pla
s=c.slice(Base.Vector(0,1,0),0)  #

# here the result is a single wire
# depending on the source object this can be several wires
s=s[0]

# if you only need the vertexes of the shape you can use
v=[]
for i in s.Vertexes:
    v.append(i.Point)

# but you can also sub-sample the section to have a certain number of points (int) ...
p1=s.discretize(20)
ii=0
for i in p1:
    ii+=1
    print i                                              # Vector()
    print ii, ": X:", i.x, " Y:", i.y, " Z:", i.z        # Vector decode
Draft.makeWire(p1,closed=False,face=False,support=None)  # to see the difference accuracy (20)

## uncomment to use
#import Draft
#Draft.downgrade(App.ActiveDocument.ActiveObject,delete=True)  # first transform the DWire in Wire
#Draft.downgrade(App.ActiveDocument.ActiveObject,delete=True)  # second split the Wire in single objects
#
##Draft.upgrade(FreeCADGui.Selection.getSelection(),delete=True) # to attach lines contiguous SELECTED us


# ... or define a sampling distance (float)
p2=s.discretize(0.5)
ii=0
```

```
for i in p2:
    ii+=1
    print i                                          # Vector()
    print ii, ": X:", i.x, " Y:", i.y, " Z:", i.z    # Vector decode
Draft.makeWire(p2,closed=False,face=False,support=None)  # to see the difference accuracy (0.5)

## uncomment to use
#import Draft
#Draft.downgrade(App.ActiveDocument.ActiveObject,delete=True)  # first transform the DWire in Wire
#Draft.downgrade(App.ActiveDocument.ActiveObject,delete=True)  # second split the Wire in single objects
#
##Draft.upgrade(FreeCADGui.Selection.getSelection(),delete=True) # to attach lines contiguous SELECTED us
```

Select all objects in the document

```
import FreeCAD
for obj in FreeCAD.ActiveDocument.Objects:
    print obj.Name                                # display the object Name
    objName = obj.Name
    obj = App.ActiveDocument.getObject(objName)
    Gui.Selection.addSelection(obj)               # select the object
```

Selecting a face of an object

```
# select one face of the object
import FreeCAD, Draft
App=FreeCAD
nameObject = "Box"                            # objet
faceSelect = "Face3"                          # face to selection
loch=App.ActiveDocument.getObject(nameObject)  # objet
Gui.Selection.clearSelection()                # clear all selection
Gui.Selection.addSelection(loch,faceSelect)   # select the face specified
s = Gui.Selection.getSelectionEx()
#Draft.makeFacebinder(s)                      #
```

Create one object to the position of the Camera

```
# create one object of the position to camera with "getCameraOrientation()"
# the object is still facing the screen
import Draft

plan = FreeCADGui.ActiveDocument.ActiveView.getCameraOrientation()
plan = str(plan)
###### extract data
a    = ""
for i in plan:
    if i in ("0123456789e.- "):
        a+=i
a = a.strip(" ")
a = a.split(" ")
####### extract data

#print a
#print a[0]
#print a[1]
#print a[2]
#print a[3]
```

```
xP = float(a[0])
yP = float(a[1])
zP = float(a[2])
qP = float(a[3])

pl = FreeCAD.Placement()
pl.Rotation.Q = (xP,yP,zP,qP)         # rotation of object
pl.Base = FreeCAD.Vector(0.0,0.0,0.0) # here coordinates XYZ of Object
rec = Draft.makeRectangle(length=10.0,height=10.0,placement=pl,face=False,support=None) # create rectang
#rec = Draft.makeCircle(radius=5,placement=pl,face=False,support=None)                   # create circle
print rec.Name
```

here same code simplified

```
import Draft
pl = FreeCAD.Placement()
pl.Rotation = FreeCADGui.ActiveDocument.ActiveView.getCameraOrientation()
pl.Base = FreeCAD.Vector(0.0,0.0,0.0)
rec = Draft.makeRectangle(length=10.0,height=10.0,placement=pl,face=False,support=None)
```

< précédent: Embedding FreeCAD Index
suivant: Line drawing function >

Cette page montre comment construire facilement des fonctionnalités avancées en Python. Dans cet exercice, nous allons construire un nouvel outil qui trace une ligne. Cet outil peut alors être lié à une commande FreeCAD, et cette commande peut être appelée par n'importe quel élément de l'interface, comme un élément de menu ou un bouton de la barre d'outils.

Script principal

Première chose, nous allons écrire un script contenant toutes nos fonctionnalités, puis, nous allons l'enregistrer dans un fichier, et l'importer dans FreeCAD, alors toutes les classes et fonctions que nous écrirons seront accessibles à partir de FreeCAD.
Alors, lancez votre éditeur de texte favori, et entrez les lignes suivantes:

```
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)
```

Explications détaillées

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

En Python, lorsque vous voulez utiliser les fonctions d'un autre module, vous avez besoin de l'importer.
Dans notre cas, nous aurons besoin de fonctions du **Part Module**, pour la création de la ligne, et du **Gui module**

(FreeCADGui), pour accéder à la vue 3D.
Nous avons également besoin de tout le contenu de la bibliothèque de pièces, afin que nous puissions utiliser directement tous les objets comme **coin**, **SoMouseButtonEvent** (évènement souris) etc ..

```
class line:
```

Ici, nous définissons notre classe principale.
Mais pourquoi utilisons-nous une classe et non une fonction ? La raison en est que nous avons besoin que notre outil reste "vivant" en attendant que l'utilisateur clique sur l'écran.

- Une fonction se termine lorsque sa tâche est terminée,
- mais un objet, **(une classe définit un objet)** reste en vie (actif) jusqu'à ce qu'il soit détruit.

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

En Python, toutes les classes ou fonctions peuvent avoir une description.
Ceci est particulièrement utile dans FreeCAD, parce que quand vous appelez cette classe dans l'interpréteur, la description sera affichée comme une **info-bulle**.

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

Les classes Python doivent toujours contenir une fonction **__init__**, qui est exécutée lorsque la classe est appelée pour créer un objet.
Donc, nous allons mettre ici tout ce que nous voulons produire lorsque notre outil de création de ligne commence (appelé).

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

Dans une classe, il est généralement souhaitable d'ajouter **self.** devant un nom de variable, de sorte que la variable sera facilement accessible à toutes les fonctions à l'intérieur et à l'extérieur de cette classe.

Ici, nous allons utiliser **self.view** pour accéder et manipuler la vue active 3D.

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

Ici, nous créons une liste vide qui contiendra les **points** en 3D envoyés par la fonction **GetPoint**.

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

Ceci est un point important:
Du fait qu'il s'agit d'une scène coin3D (http://www.coin3d.org/), FreeCAD utilise les mécanismes de rappel de **coin**, qui permet à une fonction d'être appelée à chaque fois qu'un évènement se passe sur la scène.
Dans notre cas, nous créons un appel pour SoMouseButtonEvent (http://doc.coin3d.org/Coin/group__events.html), et nous le lions à la fonction **GetPoint**.
Maintenant, chaque fois qu'un bouton de la souris est enfoncé ou relâché, la fonction **GetPoint** sera exécutée.

Notez qu'il existe aussi une alternative à **addEventCallbackPivy()** appelée **addEventCallback()** qui dispense l'utilisation de **pivy**. Mais, **pivy** est un moyen très simple et efficace d'accéder à n'importe quelle partie de la scène **coin**, il est conseillé de l'utiliser autant que possible !

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

Maintenant, nous définissons la fonction **GetPoint**, qui sera exécutée quand un bouton de la souris sera pressé dans une vue 3D.
Cette fonction recevra un argument, que nous appellerons **event_cb**. A partir de l'appel de cet événement, nous pouvons accéder à l'objet événement, qui contient plusieurs éléments d'information (plus d'informations sur cette page (http://www.freecadweb.org/wiki/index.php?title=Code_snippets/fr#Observation_des_.C3.A9v.C3.A8nements_de_la_souris_dans_la_

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

La fonction **GetPoint** sera appelée dès qu'un bouton de la souris est enfoncé ou relâché. Mais, nous ne voulons prendre un point 3D uniquement lorsqu'il est pressé (sinon, nous aurons deux points 3D très proches l'un de l'autre).
Donc, nous devons vérifier cela avec:

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

Ici, nous avons les coordonnées du curseur de la souris sur l'écran

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

Cette fonction nous donne le vecteur (**x, y, z**) du point qui se trouve sur le plan focal, juste sous curseur de notre souris.
Si vous êtes dans la vue caméra, imaginez un rayon provenant de la caméra, en passant par le curseur de la souris, et en appuyant sur le plan focal.
C'est notre point dans la vue 3D. Si l'on est en mode orthogonal, le rayon est parallèle à la direction de la vue.

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

Nous ajoutons notre nouveau point sur la pile

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

Avons nous tous les points ? si oui, alors nous allons tracer la ligne !

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

Ici, nous utilisons la fonction **line()** de Part Module qui crée une ligne de deux vecteurs FreeCAD.
Tout ce que nous créons et modifions l'intérieur de **Part Module**, reste dans le **Part Module**.

Donc, jusqu'à présent, nous avons créé une **Line Part**. Il n'est lié à aucun objet de notre document actif, c'est pour cela que rien ne s'affiche sur l'écran.

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

Le document FreeCAD ne peut accepter que des formes à partir de Part Module. Les formes sont le type le plus courant de Part Module.
Donc, nous devons transformer notre ligne en une forme avant de l'ajouter au document.

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

Le Part module a une fonction très pratique **show()** qui crée un nouvel objet dans le document et se lie a une forme.
Nous aurions aussi pu créer un nouvel objet dans le premier document, puis le lier à la forme manuellement.

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

Maintenant, nous en avons fini avec notre ligne, nous allons supprimer le mécanisme de rappel, qui consomme de précieux cycles de CPU.

Tester et utiliser un script

Maintenant, nous allons enregistrer notre script dans un endroit où l'interpréteur Python de FreeCAD le trouvera.
Lors de l'importation de modules, l'interpréteur cherchera dans les endroits suivants:

- les chemins d'installation de python,
- le répertoire bin FreeCAD,
- et tous les répertoires des modules FreeCAD.

Donc, la meilleure solution est de créer un nouveau répertoire dans le répertoire Mod de FreeCAD , et sauver votre script dans ce répertoire.

Par exemple, nous allons créer un répertoire **"myscripts"**, et sauver notre script comme **"exercise.py"**.

Maintenant, tout est prêt, nous allons commencer par créez un nouveau document FreeCAD, et, dans l'interpréteur Python, tapons:

```
import exercise
```

Si aucun message d'erreur n'apparaît, cela signifie que notre script **exercise** a été chargé.
Nous pouvons maintenant lister son contenu avec:

```
dir(exercise)
```

La commande **dir()** est une commande intégrée dans python, et lister le contenu d'un module. Nous pouvons voir que notre **classe line()** est là qui nous attend.
Maintenant, nous allons le tester:

```
exercise.line()
```

Puis, cliquez deux fois dans la vue 3D, et bingo, voici notre ligne ! Pour la faire de nouveau, tapez juste exercise.line(), encore et encore, et encore ... C'est bien, non?

Enregistrement du script dans l'interface de FreeCAD

Maintenant, pour que notre outil de création de ligne soit vraiment cool, il devrait y avoir un bouton sur l'interface, nous n'aurons donc pas besoin de taper tout ce code à chaque fois.
Le plus simple est de transformer notre nouveau répertoire **myscripts** dans un plan de travail FreeCAD. C'est facile, tout ce qui est nécessaire de faire, est de mettre un fichier appelé **InitGui.py** à l'intérieur de votre répertoire **myscripts**.
Le fichier **InitGui.py** contiendra les instructions pour créer un nouveau plan de travail, et s'ajoutera notre nouvel outil.

Sans oublier, que nous aurons aussi besoin de transformer un peu notre code **exercise**, de sorte que l'outil **line()** soit reconnu comme une commande FreeCAD officielle.
Commençons par faire un fichier **InitGui.py**, et écrivons le code suivant à l'intérieur:

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

Actuellement, vous devriez comprendre le script ci-dessus par vous-même, du moins, je pense:
Nous créons une nouvelle classe que nous appelons **MyWorkbench**, nous lui donnons un nom (MenuText), et nous définissons une fonction **Initialize()** qui sera exécutée quand le plan de travail sera chargé dans FreeCAD.
Dans cette fonction, nous chargeons le contenus de notre fichier '**exercise**, et ajoutons les commandes FreeCAD trouvées dans une liste de commandes. Ensuite, nous faisons une barre d'outils appelée "**Mes scripts**" et nous attribuons notre liste des commandes.

Actuellement, bien sûr, nous n'avons qu'un seul outil, puisque notre liste de commandes ne contient qu'un seul élément. Puis, une fois que notre plan de travail est prêt, nous l'ajoutons à l'interface principale.

Mais, cela ne fonctionne toujours pas, car une commande FreeCAD doit être formatée d'une certaine façon pour travailler. Nous aurons donc besoin de transformer un peu notre outil **ligne()**.
Notre nouveau script **exercise.py** va maintenant ressembler à ceci:

```
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 b
FreeCADGui.addCommand('line', line())
```

Qu'avons fait ici ? nous avons transformé notre fonction **__ init__ ()** en une fonction **Activated()**, parce que lorsque les commandes sont exécutées dans FreeCAD, il exécute automatiquement la fonction **Activated()**.
Nous avons également ajouté une fonction **GetResources()**, qui informe FreeCAD où se trouve l'icône de l'outil, le nom et l'info-bulle de l'outil.
Toute image, jpg, png ou svg peut être utilisé comme icône, il peut être de n'importe quelle taille, mais il est préférable d'utiliser une taille standard qui est proche de l'aspect final, comme 16x16, 24x24 ou 32x32.
Puis, nous ajoutons notre **class line()** comme une commande officielle de FreeCAD avec la méthode **addCommand()**.

Ça y est, nous avons juste besoin de redémarrer FreeCAD et nous aurons un plan de travail agréable avec notre nouvel outil **ligne** tout neuf !

Vous voulez en savoir plus ?

Si vous avez aimé cet **"exercise"**, pourquoi ne pas essayer d'améliorer ce petit outil ? Il y a beaucoup de choses à faire, comme par exemple:

- Ajouter des Commentaires utilisateur: jusqu'à présent nous avons fait un outil très dépouillé, l'utilisateur peut être un peu perdu lors de son utilisation. Vous pouvez ajouter vos commentaires, en guidant l'utilisateur. Par exemple, vous pourriez émettre des messages à la console FreeCAD. "Jetez"

un oeil dans le module **FreeCAD.Console**
- Ajouter la possibilité d'entrer les coordonnées 3D manuellement . Regardez les fonctions Python input(), par exemple
- Ajouter la possibilité d'ajouter plus de 2 points
- Ajouter des événements pour d'autres fonctions: Maintenant que nous venons d'apprendre les événements de bouton de souris, si nous souhaitons également faire quelque chose quand la souris est déplacée, comme par exemple l'affichage des coordonnées actuelles?
- Donnez un nom à l'objet créé et bien d'autres choses

N'hésitez pas de commenter vos idées ou questions sur le forum (http://forum.freecadweb.org/) !

< précédent: Code snippets Index suivant: Dialog creation >

Dans cette page nous allons vous montrer comment construire une simple boîte de dialogue avec Qt Designer (http://qt-project.org/doc/qt-4.8/designer-manual.html), Qt Designer, est l'outil officiel de Qt pour la conception d'interfaces (Gui), puis de le convertir en code Python, et l'utiliser à l'intérieur de FreeCAD. Je vais supposer, que pour l'exemple, vous savez déjà comment modifier et exécuter un script Python, et que vous pouvez travailler avec des choses simples dans une fenêtre de terminal tel que se déplacer, etc . . Bien sûr, vous devez également avoir installé **PySide**.

Construire une boîte de dialogue

Dans les applications de CAO, bien concevoir une **UI** (interface utilisateur) est très important.
Tout ce que l'utilisateur fera, se fera à travers un outil de l'interface: la lecture des boîtes de dialogue, appuyer sur les boutons, le choix entre les icônes, etc . .
Il est donc très important de réfléchir attentivement à la conception de votre boîte de dialogue, comment vous voulez que l'utilisateur se comporter avec la boîte, et comment sera le flux de travail de votre action.

Il y a une deux choses à savoir lors de la conception de l'interface:

- Boîtes de dialogue modales ou non-modale (http://fr.wikipedia.org/wiki/Fenêtre_modale) :
  - Une boîte de dialogue **modale** apparaît en face de votre écran et, arrête l'action de la fenêtre principale, forçant l'utilisateur à répondre à la boîte de dialogue.
  - Une boîte de dialogue **non modale** ne vous empêche pas de travailler sur la Fenêtre principale, vous pouvez travailler sur les deux fenêtres.

Dans certains cas, le premier est préférable, dans d'autres cas non.

- Identifier ce qui est nécessaire et ce qui est optionnel:
  - Assurez-vous que l'utilisateur sait ce qu'il doit faire.

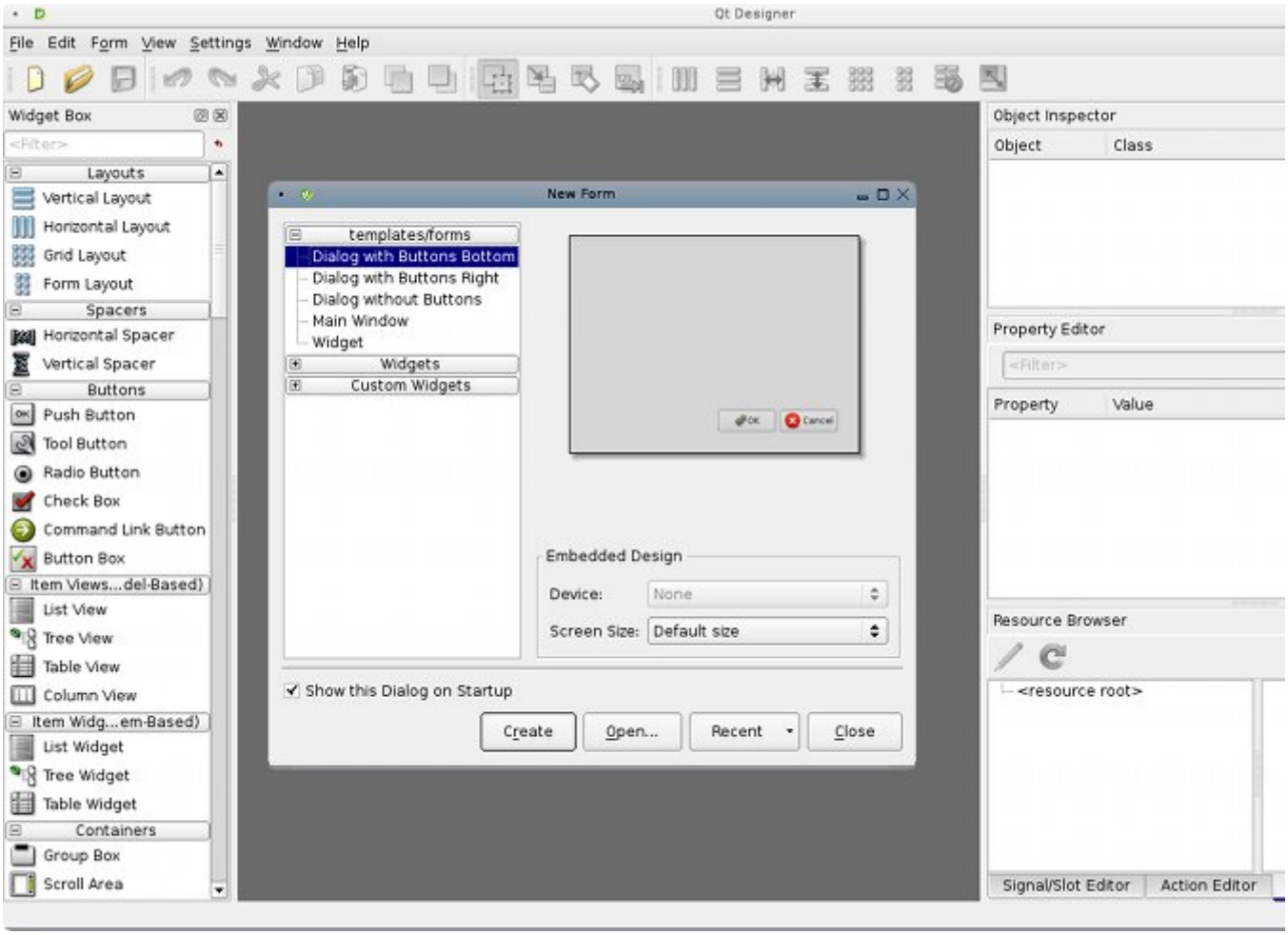
Prévoyez des étiquettes avec des descriptions appropriées, des info-bulles d'utilisation, etc . .

- Séparez les commandes à partir de paramètres:
  - Cela se fait habituellement avec des boutons et des champs de saisie de texte.
  - L'utilisateur sait que cliquer sur un bouton va produire une action, tout en changeant une valeur dans un champ de texte, va changer un paramètre quelque part. Cependant, aujourd'hui, les utilisateurs savent généralement bien ce qu'est un bouton, ce qu'est un champ de saisie, etc . . .

La boîte à outils de l'interface **Qt** que nous utilisons, est une boîte à outils **state-of-the-art** (interface graphique avancée), et nous n'aurons pas beaucoup d'inquiétudes pour rendre les choses claires, car elles sont déjà très claires par elles-mêmes.

Donc, maintenant que nous avons bien défini ce que nous ferons, il est temps d'ouvrir **Qt Designer**.
Nous allons concevoir très facilement une simple boîte de dialogue, comme ceci:

Nous allons ensuite utiliser cette boîte de dialogue dans FreeCAD pour produire une belle surface plane rectangulaire.
Vous ne trouverez peut-être pas très utile de produire de beaux plans rectangulaires, mais il sera facile de le changer plus tard et de faire des choses plus complexes.
Lorsque vous l'ouvrez, Qt Designer ressemble à ceci:



Il est très simple à utiliser. Sur la barre de gauche vous avez des éléments qui peuvent être glissés sur votre widget (tous les outils). Sur le côté droit vous avez des panneaux d'affichage de propriétés de toutes sortes, des propriétés de certains éléments modifiables.
Donc, commencez par créer un nouveau widget. Sélectionnez "**Dialog without buttons**", car nous ne voulons pas de boutons par défaut Ok/Annuler. Ensuite, faites glisser sur votre widget **3 labels**, un pour le titre, un pour l'écriture "Height" (Hauteur) et l'autre pour l'écriture "Width" (Largeur).

Les **labels** (étiquettes) sont de simples textes qui apparaissent sur votre widget, il servent a informer l'utilisateur.
Si vous sélectionnez un **label**, sur le côté droit apparaîssent plusieurs propriétés que vous pouvez modifier, comme le style de police, taille, etc . . .

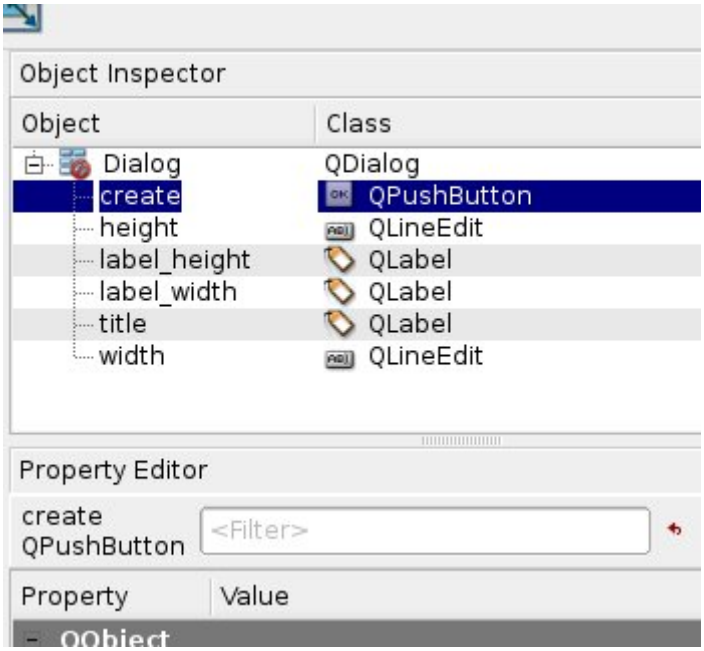
Ensuite, ajoutez 2 **LineEdits** , qui sont des champs texte que l'utilisateur peut remplir, un pour la hauteur et l'autre pour la

largeur.

Ici aussi, nous pouvons modifier les propriétés. Par exemple, pourquoi ne pas définir une valeur par défaut ? Par exemple 1,00 pour chacun d'eux.
De cette façon, lorsque l'utilisateur verra la boîte de dialogue, les deux valeurs seront déjà remplies et si les valeurs conviennent, il peut directement appuyer sur le bouton, gain de temps précieux.
Ensuite, ajoutez un **PushButton** , qui est le bouton, que l'utilisateur devra appuyer après avoir rempli les 2 champs.

Notez que j'ai choisi ici des contrôles très simples, mais **Qt** a beaucoup plus d'options, par exemple, vous pouvez utiliser **spinbox** au lieu de **LineEdits**, etc ..
Regardez tout ce qui est disponible, vous aurez sûrement d'autres idées.

C'est à peu près tout ce que nous devons faire dans Qt Designer.
Une dernière chose, nous allons renommer tous nos éléments avec des noms faciles, de sorte qu'il sera plus facile de les identifier dans nos scripts:



Conversion de notre boîte de dialogue en code Python avec "pyuic"

Maintenant, nous allons sauver notre widget quelque part. Il sera sauvegardé dans un fichier **.Ui**, que nous allons facilement convertir en script Python avec **pyuic**.
Dans windows, le programme est livré avec **pyuic pyqt** (à vérifier), sur Linux, vous aurez probablement besoin de l'installer séparément à partir de votre gestionnaire de paquets (sur debian-systèmes basés sur, il fait partie du paquet pyqt4-dev-tools).
Pour faire la conversion, vous aurez besoin d'ouvrir une fenêtre de terminal (ou une fenêtre d'invite de commandes), accédez à l'endroit où vous avez enregistré votre fichier **ui** :

- **pyuic.py** est l'outil qui convertit les fichiers qt-designer **.ui** (Interface Utilisateur) en fichier **.py** (code Python), la ligne de commande dans la console DOS est :

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

- vous pouvez créer un fichier **.bat** pour automatiser la commande:
- copiez cette ligne dans un fichier texte et sauvez le sous le nom **"compile.bat"**

```
@"C:\Python27\python" "C:\Python27\Lib\site-packages\PyQt4\uic\pyuic.py" -x %1.ui > %1.py
```

- puis tapez à la ligne de commande " **compile fichier** " sans extension, le nom "**fichier**" entré **.ui**, sera le nom sortant avec extension **.py**
- **ATTENTION: il faut que les fichiers soient présents et accessibles, vérifiez que les fichiers sont présents et que les chemins sont justes !**
- pour cet exemple entièrement automatique et simplifié, **"compile.bat"** est au même endroit que le **fichier.ui** à convertir en **fichier.py**

Autres liens de documentation "Python and Qt" (http://www.qtrac.eu/pyqtbook.html) , sur Développez.com (http://ogirardot.developpez.com/introduction-pyqt/) et bien d'autres.

Sur certains systèmes, le programme est appelé **pyuic4** au lieu de **pyuic** (attention à la compatibilité). Il sert simplement de convertisseur de fichier **.Ui** en un script python **.py**.

Si nous ouvrons le fichier mywidget.py, son contenu est très facile à comprendre:

```
from PySide import QtCore, QtGui

class Ui_Dialog(object):
    def setupUi(self, Dialog):
        Dialog.setObjectName("Dialog")
        Dialog.resize(187, 178)
        self.title = QtGui.QLabel(Dialog)
        self.title.setGeometry(QtCore.QRect(10, 10, 271, 16))
        self.title.setObjectName("title")
        self.label_width = QtGui.QLabel(Dialog)
        ...

        self.retranslateUi(Dialog)
        QtCore.QMetaObject.connectSlotsByName(Dialog)

   def retranslateUi(self, Dialog):
        Dialog.setWindowTitle(QtGui.QApplication.translate("Dialog", "Dialog", None, QtGui.QApplication.
        self.title.setText(QtGui.QApplication.translate("Dialog", "Plane-O-Matic", None, QtGui.QApplicat
        ...
```

Comme vous voyez, il a une structure très simple: une classe nommée **Ui_Dialog** est créé, qui stocke les éléments de l'interface de notre widget.
Cette classe dispose de deux méthodes, une pour la mise en place du widget, et l'autre pour traduire son contenu, qui fait partie du mécanisme général de Qt pour la traduction des éléments d'interface.
La méthode de configuration, crée simplement, un par un, les widgets tels que nous les avons définis dans Qt Designer, et définit leurs options aussi comme nous avons décidé plus tôt.

Puis, toute l'interface est traduite, et enfin, les "slots" se connectent (nous en reparlerons plus tard).

Nous pouvons maintenant créer un nouveau widget, et utiliser cette classe pour créer son interface.
Nous pouvons déjà voir notre widget en action, en mettant notre fichier mywidget.py dans un endroit où FreeCAD la trouvera (dans le répertoire bin FreeCAD, ou dans l'un des sous-répertoires Mod), et, dans l'interpréteur Python de FreeCAD, faisons:

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

Et notre boîte de dialogue apparaîtra! Notez que notre interpréteur Python fonctionne toujours, nous avons une boîte de dialogue non modale.
Donc, pour la fermer, nous pouvons (à part cliquer sur son icône, bien sûr) faire:

```
d.hide()
```

Faire quelque chose avec notre boîte de dialogue

Maintenant que nous pouvons afficher et masquer notre boîte de dialogue, nous avons juste besoin d'ajouter la dernière partie, pour en faire quelque chose !
Si vous explorez un peu Qt Designer, vous découvrirez rapidement toute une section appelée "**signaux et slots**".
Fondamentalement, cela fonctionne comme ceci, ce sont les éléments sur vos widgets (dans la terminologie de Qt, ces éléments sont eux-mêmes des widgets) qui peuvent envoyer des signaux.

Ces signaux diffèrent selon le type de widget. Par exemple, un bouton peut envoyer un signal quand il est pressé et quand il est relâché.
Ces signaux peuvent être connectés à des créneaux, qui peuvent être des fonctionnalités spéciales d'autres widgets (par exemple une boîte de dialogue a un bouton "Fermer" sur lequel vous pouvez connecter le signal à partir d'un autre bouton "Fermer"), ou, peuvent être des fonctions personnalisées.
La documentation de référence PyQt (http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/classes.html) répertorie tous les widgets Qt, ce qu'ils peuvent faire, ce qu'ils signalent, ce qu'ils peuvent envoyer, etc . .

.

Ce que nous allons faire ici, c'est créer une nouvelle fonction qui permettra de créer une surface plane basée sur la hauteur et la largeur, et, relier cette fonction au bouton "Create!".
Donc, nous allons commencer par importer nos modules FreeCAD, en mettant la ligne suivante en haut du script, où nous importons déjà **QtCore** et **QtGui**:

```
import FreeCAD, Part
```

Ensuite, nous allons ajouter une nouvelle fonction à notre classe **Ui_Dialog**:

```
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()
```

Puis, nous avons besoin d'informer Qt pour qu'il se connecte sur le bouton de la fonction, en plaçant la ligne suivante juste avant **QtCore.QMetaObject.connectSlotsByName(Dialog)**:

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

Il s'agit, comme vous le voyez, de relier le signal du bouton enfoncé de l'objet a créer (**"Create!" Bouton**), à un emplacement nommé **createPlane**, dont nous venons de définir.
Ça y est ! Maintenant, la touche finale, nous pouvons ajouter une petite fonction, pour créer la boîte de dialogue, elle sera plus facile a appeler.
En dehors de la classe **Ui_Dialog class**, nous allons ajouter le code suivant:

```
class plane():
   def __init__(self):
       self.d = QtGui.QWidget()
       self.ui = Ui_Dialog()
       self.ui.setupUi(self.d)
       self.d.show()
```

(Rappel sur Python : la méthode **__init__** est une classe qui s'exécute automatiquement chaque fois qu'un nouvel objet est créé !)

Puis, à partir de FreeCAD, nous avons seulement besoin de faire:

```
import mywidget
myDialog = mywidget.plane()
```

Voilà, c'est tout ...
Maintenant, vous pouvez essayer toutes sortes de choses, comme par exemple l'insertion de votre widget dans l'interface FreeCAD (voir la page Code snippets), ou, faire des outils personnalisés beaucoup plus avancés, en utilisant d'autres éléments dans votre widget.

Le script complet

Ceci est le script de référence complet:

```
# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file 'mywidget.ui'
#
# Created: Mon Jun  1 19:09:10 2009
#      by: PyQt4 UI code generator 4.4.4
# Modified for PySide 16:02:2015
# WARNING! All changes made in this file will be lost!

from PySide import QtCore, QtGui
import FreeCAD, Part

class Ui_Dialog(object):
   def setupUi(self, Dialog):
       Dialog.setObjectName("Dialog")
       Dialog.resize(187, 178)
       self.title = QtGui.QLabel(Dialog)
       self.title.setGeometry(QtCore.QRect(10, 10, 271, 16))
       self.title.setObjectName("title")
       self.label_width = QtGui.QLabel(Dialog)
       self.label_width.setGeometry(QtCore.QRect(10, 50, 57, 16))
       self.label_width.setObjectName("label_width")
       self.label_height = QtGui.QLabel(Dialog)
       self.label_height.setGeometry(QtCore.QRect(10, 90, 57, 16))
       self.label_height.setObjectName("label_height")
       self.width = QtGui.QLineEdit(Dialog)
       self.width.setGeometry(QtCore.QRect(60, 40, 111, 26))
```

```
        self.width.setObjectName("width")
        self.height = QtGui.QLineEdit(Dialog)
        self.height.setGeometry(QtCore.QRect(60, 80, 111, 26))
        self.height.setObjectName("height")
        self.create = QtGui.QPushButton(Dialog)
        self.create.setGeometry(QtCore.QRect(50, 140, 83, 26))
        self.create.setObjectName("create")

        self.retranslateUi(Dialog)
        QtCore.QObject.connect(self.create,QtCore.SIGNAL("pressed()"),self.createPlane)
        QtCore.QMetaObject.connectSlotsByName(Dialog)

    def retranslateUi(self, Dialog):
        Dialog.setWindowTitle(QtGui.QApplication.translate("Dialog", "Dialog", None, QtGui.QApplication.Ui
        self.title.setText(QtGui.QApplication.translate("Dialog", "Plane-O-Matic", None, QtGui.QApplicatio
        self.label_width.setText(QtGui.QApplication.translate("Dialog", "Width", None, QtGui.QApplication.
        self.label_height.setText(QtGui.QApplication.translate("Dialog", "Height", None, QtGui.QApplicatio
        self.create.setText(QtGui.QApplication.translate("Dialog", "Create!", None, QtGui.QApplication.Un

    def createPlane(self):
        try:
            # first we check if valid numbers have been entered
            w = float(self.width.text())
            h = float(self.height.text())
        except ValueError:
            print "Error! Width and Height values must be valid numbers!"
        else:
            # create a face from 4 points
            p1 = FreeCAD.Vector(0,0,0)
            p2 = FreeCAD.Vector(w,0,0)
            p3 = FreeCAD.Vector(w,h,0)
            p4 = FreeCAD.Vector(0,h,0)
            pointslist = [p1,p2,p3,p4,p1]
            mywire = Part.makePolygon(pointslist)
            myface = Part.Face(mywire)
            Part.show(myface)

class plane():
    def __init__(self):
        self.d = QtGui.QWidget()
        self.ui = Ui_Dialog()
        self.ui.setupUi(self.d)
        self.d.show()
```

Création d'une boîte de dialogue avec ses boutons

Méthode 1

Un exemple d'une boîte de dialogue complète avec ses connections.

```
# -*- coding: utf-8 -*-
# Create by flachyjoe

from PySide import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    def _fromUtf8(s):
        return s
```

```
try:
    _encoding = QtGui.QApplication.UnicodeUTF8
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig, _encoding)
except AttributeError:
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig)


class Ui_MainWindow(object):

    def __init__(self, MainWindow):
        self.window = MainWindow

        MainWindow.setObjectName(_fromUtf8("MainWindow"))
        MainWindow.resize(400, 300)
        self.centralWidget = QtGui.QWidget(MainWindow)
        self.centralWidget.setObjectName(_fromUtf8("centralWidget"))

        self.pushButton = QtGui.QPushButton(self.centralWidget)
        self.pushButton.setGeometry(QtCore.QRect(30, 170, 93, 28))
        self.pushButton.setObjectName(_fromUtf8("pushButton"))
        self.pushButton.clicked.connect(self.on_pushButton_clicked) #connection pushButton

        self.lineEdit = QtGui.QLineEdit(self.centralWidget)
        self.lineEdit.setGeometry(QtCore.QRect(30, 40, 211, 22))
        self.lineEdit.setObjectName(_fromUtf8("lineEdit"))
        self.lineEdit.returnPressed.connect(self.on_lineEdit_clicked) #connection lineEdit

        self.checkBox = QtGui.QCheckBox(self.centralWidget)
        self.checkBox.setGeometry(QtCore.QRect(30, 90, 81, 20))
        self.checkBox.setChecked(True)
        self.checkBox.setObjectName(_fromUtf8("checkBoxON"))
        self.checkBox.clicked.connect(self.on_checkBox_clicked) #connection checkBox

        self.radioButton = QtGui.QRadioButton(self.centralWidget)
        self.radioButton.setGeometry(QtCore.QRect(30, 130, 95, 20))
        self.radioButton.setObjectName(_fromUtf8("radioButton"))
        self.radioButton.clicked.connect(self.on_radioButton_clicked) #connection radioButton

        MainWindow.setCentralWidget(self.centralWidget)

        self.menuBar = QtGui.QMenuBar(MainWindow)
        self.menuBar.setGeometry(QtCore.QRect(0, 0, 400, 26))
        self.menuBar.setObjectName(_fromUtf8("menuBar"))
        MainWindow.setMenuBar(self.menuBar)

        self.mainToolBar = QtGui.QToolBar(MainWindow)
        self.mainToolBar.setObjectName(_fromUtf8("mainToolBar"))
        MainWindow.addToolBar(QtCore.Qt.TopToolBarArea, self.mainToolBar)

        self.statusBar = QtGui.QStatusBar(MainWindow)
        self.statusBar.setObjectName(_fromUtf8("statusBar"))
        MainWindow.setStatusBar(self.statusBar)

        self.retranslateUi(MainWindow)

    def retranslateUi(self, MainWindow):
        MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow", None))
        self.pushButton.setText(_translate("MainWindow", "OK", None))
        self.lineEdit.setText(_translate("MainWindow", "tyty", None))
        self.checkBox.setText(_translate("MainWindow", "CheckBox", None))
        self.radioButton.setText(_translate("MainWindow", "RadioButton", None))

    def on_checkBox_clicked(self):
        if self.checkBox.checkState()==0:
            App.Console.PrintMessage(str(self.checkBox.checkState())+"  CheckBox KO\r\n")
        else:
            App.Console.PrintMessage(str(self.checkBox.checkState())+" CheckBox OK\r\n")
#        App.Console.PrintMessage(str(self.lineEdit.setText("tititi"))+" LineEdit\r\n") #write text to t
#        str(self.lineEdit.setText("tititi")) #écrit le texte dans la fenêtre lineEdit
        App.Console.PrintMessage(str(self.lineEdit.displayText())+" LineEdit\r\n")

    def on_radioButton_clicked(self):
        if self.radioButton.isChecked():
```

```
            App.Console.PrintMessage(str(self.radioButton.isChecked())+" Radio OK\r\n")
        else:
            App.Console.PrintMessage(str(self.radioButton.isChecked())+"  Radio KO\r\n")

    def on_lineEdit_clicked(self):
#        if self.lineEdit.textChanged():
                App.Console.PrintMessage(str(self.lineEdit.displayText())+" LineEdit Display\r\n")

    def on_pushButton_clicked(self):
        App.Console.PrintMessage("Terminé\r\n")
        self.window.hide()

MainWindow = QtGui.QMainWindow()
ui = Ui_MainWindow(MainWindow)
MainWindow.show()
```

Ici la même fenêtre mais avec un icône sur chaque bouton.

```
# -*- coding: utf-8 -*-

from PySide import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    def _fromUtf8(s):
        return s

try:
    _encoding = QtGui.QApplication.UnicodeUTF8
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig, _encoding)
except AttributeError:
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig)


class Ui_MainWindow(object):

    def __init__(self, MainWindow):
        self.window = MainWindow
#        path = FreeCAD.ConfigGet("UserAppData")
        path = FreeCAD.ConfigGet("AppHomePath")

        MainWindow.setObjectName(_fromUtf8("MainWindow"))
        MainWindow.resize(400, 300)
        self.centralWidget = QtGui.QWidget(MainWindow)
        self.centralWidget.setObjectName(_fromUtf8("centralWidget"))

        self.pushButton = QtGui.QPushButton(self.centralWidget)
        self.pushButton.setGeometry(QtCore.QRect(30, 170, 93, 28))
        self.pushButton.setObjectName(_fromUtf8("pushButton"))
        self.pushButton.clicked.connect(self.on_pushButton_clicked) #connection pushButton

        self.lineEdit = QtGui.QLineEdit(self.centralWidget)
        self.lineEdit.setGeometry(QtCore.QRect(30, 40, 211, 22))
        self.lineEdit.setObjectName(_fromUtf8("lineEdit"))
        self.lineEdit.returnPressed.connect(self.on_lineEdit_clicked) #connection lineEdit

        self.checkBox = QtGui.QCheckBox(self.centralWidget)
        self.checkBox.setGeometry(QtCore.QRect(30, 90, 100, 20))
        self.checkBox.setChecked(True)
        self.checkBox.setObjectName(_fromUtf8("checkBoxON"))
        self.checkBox.clicked.connect(self.on_checkBox_clicked) #connection checkBox

        self.radioButton = QtGui.QRadioButton(self.centralWidget)
        self.radioButton.setGeometry(QtCore.QRect(30, 130, 95, 20))
        self.radioButton.setObjectName(_fromUtf8("radioButton"))
        self.radioButton.clicked.connect(self.on_radioButton_clicked) #connection radioButton

        MainWindow.setCentralWidget(self.centralWidget)
```

```
        self.menuBar = QtGui.QMenuBar(MainWindow)
        self.menuBar.setGeometry(QtCore.QRect(0, 0, 400, 26))
        self.menuBar.setObjectName(_fromUtf8("menuBar"))
        MainWindow.setMenuBar(self.menuBar)

        self.mainToolBar = QtGui.QToolBar(MainWindow)
        self.mainToolBar.setObjectName(_fromUtf8("mainToolBar"))
        MainWindow.addToolBar(QtCore.Qt.TopToolBarArea, self.mainToolBar)

        self.statusBar = QtGui.QStatusBar(MainWindow)
        self.statusBar.setObjectName(_fromUtf8("statusBar"))
        MainWindow.setStatusBar(self.statusBar)

        self.retranslateUi(MainWindow)

        # Affiche un icône sur le bouton PushButton
        # self.image_01 = "C:\Program Files\FreeCAD0.13\icone01.png" # adapt the icon name
        self.image_01 = path+"icone01.png" # adapt the name of the icon
        icon01 = QtGui.QIcon()
        icon01.addPixmap(QtGui.QPixmap(self.image_01),QtGui.QIcon.Normal, QtGui.QIcon.Off)
        self.pushButton.setIcon(icon01)
        self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the direction c

        # Affiche un icône sur le bouton RadioButton
        # self.image_02 = "C:\Program Files\FreeCAD0.13\icone02.png" # adapt the name of the icon
        self.image_02 = path+"icone02.png" # adapter le nom de l'icône
        icon02 = QtGui.QIcon()
        icon02.addPixmap(QtGui.QPixmap(self.image_02),QtGui.QIcon.Normal, QtGui.QIcon.Off)
        self.radioButton.setIcon(icon02)
        # self.radioButton.setLayoutDirection(QtCore.Qt.RightToLeft) #  This command reverses the direct

        # Affiche un icône sur le bouton CheckBox
        # self.image_03 = "C:\Program Files\FreeCAD0.13\icone03.png" # the name of the icon
        self.image_03 = path+"icone03.png" # adapter le nom de l'icône
        icon03 = QtGui.QIcon()
        icon03.addPixmap(QtGui.QPixmap(self.image_03),QtGui.QIcon.Normal, QtGui.QIcon.Off)
        self.checkBox.setIcon(icon03)
        # self.checkBox.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the direction c


    def retranslateUi(self, MainWindow):
        MainWindow.setWindowTitle(_translate("MainWindow", "FreeCAD", None))
        self.pushButton.setText(_translate("MainWindow", "OK", None))
        self.lineEdit.setText(_translate("MainWindow", "tyty", None))
        self.checkBox.setText(_translate("MainWindow", "CheckBox", None))
        self.radioButton.setText(_translate("MainWindow", "RadioButton", None))

    def on_checkBox_clicked(self):
        if self.checkBox.checkState()==0:
            App.Console.PrintMessage(str(self.checkBox.checkState())+"  CheckBox KO\r\n")
        else:
            App.Console.PrintMessage(str(self.checkBox.checkState())+" CheckBox OK\r\n")
           # App.Console.PrintMessage(str(self.lineEdit.setText("tititi"))+" LineEdit\r\n") # write text
           # str(self.lineEdit.setText("tititi")) #écrit le texte dans la fenêtre lineEdit
        App.Console.PrintMessage(str(self.lineEdit.displayText())+" LineEdit\r\n")

    def on_radioButton_clicked(self):
        if self.radioButton.isChecked():
             App.Console.PrintMessage(str(self.radioButton.isChecked())+" Radio OK\r\n")
        else:
             App.Console.PrintMessage(str(self.radioButton.isChecked())+"  Radio KO\r\n")

    def on_lineEdit_clicked(self):
         # if self.lineEdit.textChanged():
         App.Console.PrintMessage(str(self.lineEdit.displayText())+" LineEdit Display\r\n")

    def on_pushButton_clicked(self):
        App.Console.PrintMessage("Terminé\r\n")
        self.window.hide()

MainWindow = QtGui.QMainWindow()
ui = Ui_MainWindow(MainWindow)
MainWindow.show()
```

ici le code pour afficher l'icône sur le **pushButton**, modifiez le nom pour un autre bouton, (**radioButton, checkBox**) ainsi que le chemin de l'icône.

```
        # Affiche un icône sur le bouton PushButton
        # self.image_01 = "C:\Program Files\FreeCAD0.13\icone01.png" # the name of the icon
        self.image_01 = path+"icone01.png" # the name of the icon
        icon01 = QtGui.QIcon()
        icon01.addPixmap(QtGui.QPixmap(self.image_01),QtGui.QIcon.Normal, QtGui.QIcon.Off)
        self.pushButton.setIcon(icon01)
        self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the direction
```

La commande **UserAppData** donne le chemin utilisateur **AppHomePath** donne le chemin d'installation de FreeCAD

```
#        path = FreeCAD.ConfigGet("UserAppData")
        path = FreeCAD.ConfigGet("AppHomePath")
```

Cette commande inverse le sens horizontal du bouton, droite à gauche

```
self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the direction of the bu
```

Méthode 2

Une autre méthode pour afficher une fenêtre, ici en créant un fichier **QtForm.py** qui renferme l'entête du programme (module appelé avec **import QtForm**), et d'un deuxième module qui renferme le code de la fenêtre tous ces accessoires, et votre code (le module appelant).

Cette méthode nécessite 2 fichiers distincts, mais permet de raccourcir votre programme, en utilisant le fichier **QtForm.py** en import. Il faut alors distribuer les deux fichiers ensemble, ils sont indissociables.

Le fichier **QtForm.py**

```
# -*- coding: utf-8 -*-
# Create by flachyjoe
from PySide import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    def _fromUtf8(s):
```

```
      return s

try:
    _encoding = QtGui.QApplication.UnicodeUTF8
    def _translate(context, text, disambig):
      return QtGui.QApplication.translate(context, text, disambig, _encoding)
except AttributeError:
   def _translate(context, text, disambig):
      return QtGui.QApplication.translate(context, text, disambig)

class Form(object):
   def __init__(self, title, width, height):
      self.window = QtGui.QMainWindow()
      self.title=title
      self.window.setObjectName(_fromUtf8(title))
      self.window.setWindowTitle(_translate(self.title, self.title, None))
      self.window.resize(width, height)

   def show(self):
      self.createUI()
      self.retranslateUI()
      self.window.show()

   def setText(self, control, text):
      control.setText(_translate(self.title, text, None))
```

Le fichier appelant, qui contient la fenêtre et votre code.

Le fichier **mon_fichier.py**

Les connections sont à faire, un bon exercice.

```
# -*- coding: utf-8 -*-
# Create by flachyjoe
from PySide import QtCore, QtGui
import QtForm

class myForm(QtForm.Form):
   def createUI(self):
      self.centralWidget = QtGui.QWidget(self.window)
      self.window.setCentralWidget(self.centralWidget)

      self.pushButton = QtGui.QPushButton(self.centralWidget)
      self.pushButton.setGeometry(QtCore.QRect(30, 170, 93, 28))
      self.pushButton.clicked.connect(self.on_pushButton_clicked)

      self.lineEdit = QtGui.QLineEdit(self.centralWidget)
      self.lineEdit.setGeometry(QtCore.QRect(30, 40, 211, 22))

      self.checkBox = QtGui.QCheckBox(self.centralWidget)
      self.checkBox.setGeometry(QtCore.QRect(30, 90, 81, 20))
      self.checkBox.setChecked(True)

      self.radioButton = QtGui.QRadioButton(self.centralWidget)
      self.radioButton.setGeometry(QtCore.QRect(30, 130, 95, 20))

   def retranslateUI(self):
      self.setText(self.pushButton, "Fermer")
      self.setText(self.lineEdit, "essai de texte")
      self.setText(self.checkBox, "CheckBox")
      self.setText(self.radioButton, "RadioButton")

   def on_pushButton_clicked(self):
      self.window.hide()

myWindow=myForm("Fenêtre de test",400,300)
myWindow.show()
```

Quelques commandes utiles

```
# Here the code to display the icon on the '''pushButton''',
# change the name to another button, ('''radioButton, checkBox''') as well as the path to the icon,

      # Displays an icon on the button PushButton
      # self.image_01 = "C:\Program Files\FreeCAD0.13\icone01.png" # he name of the icon
      self.image_01 = path+"icone01.png" # the name of the icon
      icon01 = QtGui.QIcon()
      icon01.addPixmap(QtGui.QPixmap(self.image_01),QtGui.QIcon.Normal, QtGui.QIcon.Off)
      self.pushButton.setIcon(icon01)
      self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the direction o


# path = FreeCAD.ConfigGet("UserAppData") # gives the user path
  path = FreeCAD.ConfigGet("AppHomePath") # gives the installation path of FreeCAD

# This command reverses the horizontal button, right to left
self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the horizontal button

# Displays an info button
self.pushButton.setToolTip(_translate("MainWindow", "Quitter la fonction", None)) # Displays an info but

# This function gives a color button
self.pushButton.setStyleSheet("background-color: red") # This function gives a color button

# This function gives a color to the text of the button
self.pushButton.setStyleSheet("color : #ff0000") # This function gives a color to the text of the button

# combinaison des deux, bouton et texte
self.pushButton.setStyleSheet("color : #ff0000; background-color : #0000ff;" ) #  combination of the two,

# replace the icon in the main window
MainWindow.setWindowIcon(QtGui.QIcon('C:\Program Files\FreeCAD0.13\View-C3P.png'))

# connects a lineEdit on execute
self.lineEdit.returnPressed.connect(self.execute) # connects a lineEdit on "def execute" after validatio
# self.lineEdit.textChanged.connect(self.execute) # connects a lineEdit on "def execute" with each keyst

# display text in a lineEdit
self.lineEdit.setText(str(val_X)) # Displays the value in the lineEdit (convert to string)

# extract the string contained in a lineEdit
 val_X = self.lineEdit.text() # extract the (string) string contained in lineEdit
 val_X = float(val_X0)        # converted the string to an floating
 val_X = int(val_X0)          # convert the string to an integer

# This code allows you to change the font and its attributes
      font = QtGui.QFont()
      font.setFamily("Times New Roman")
      font.setPointSize(10)
      font.setWeight(10)
      font.setBold(True) # same result with tags "<b>your text</b>" (in quotes)
      self.label_6.setFont(font)
      self.label_6.setObjectName("label_6")
      self.label_6.setStyleSheet("color : #ff0000") # This function gives a color to the text
      self.label_6.setText(_translate("MainWindow", "Select a view", None))
```

En utilisant les caractères accentués, dans le cas ou vous obtenez les erreurs suivantes :

plusieurs méthodes sont possibles.

UnicodeDecodeError: 'utf8' codec can't decode bytes in

position 0-2: invalid data

```
# conversion from a lineEdit
App.activeDocument().CopyRight.Text = str(unicode(self.lineEdit_20.text() , 'ISO-8859-1').encode('UTF-8'
DESIGNED_BY = unicode(self.lineEdit_01.text(), 'ISO-8859-1').encode('UTF-8')
```

ou avec la procédure

```
def utf8(unio):
    return unicode(unio).encode('UTF8')
```

UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 9: ordinal not in range(128)

```
# conversion
a = u"Nom de l'élément : "
f.write('''a.encode('iso-8859-1')'''+str(element_)+"\n")
```

ou avec la procédure

```
def iso8859(encoder):
    return unicode(encoder).encode('iso-8859-1')
```

ou

```
iso8859(unichr(176))
```

ou

```
unichr(ord(176))
```

ou

```
uniteSs = "mm"+iso8859(unichr(178))
print unicode(uniteSs, 'iso8859')
```

< précédent: Line drawing function Index suivant: Licence >

Développer une application pour FreeCAD

Déclaration du fondateur

Je sais que la discussion sur le *« droit »* de licence pour l'open source a occupé une partie importante de la bande passante Internet alors voici la raison pour laquelle, à mon avis, FreeCAD doit être sous licence LGPL.

J'ai choisi les licences LGPL (http://fr.wikipedia.org /wiki/Licence_publique_g%C3%A9n%C3%A9rale_limit %C3%A9e_GNU) et GPL (http://fr.wikipedia.org /wiki/Licence_publique_g%C3%A9n%C3%A9rale_GNU) pour le projet, je sais qu'il y a des pros et des anti LGPL et je vous donnerai quelques raisons de cette décision.

FreeCAD est le mélange d'une bibliothèque et d'une application, de sorte que le GPL serait un peu fort pour cela. Il permettrait d'éviter l'écriture de modules commerciaux pour FreeCAD car elle empêcherait la liaison avec les librairies de base FreeCAD. Vous pouvez vous demander pourquoi des modules commerciaux ? Linux aurait-il autant de succès si les bibliothèques C GNU étaient sous licences GPL, et empêchaient donc les liaisons avec des applications non GPL ? Et bien que j'aime la liberté de Linux, je veux aussi être en mesure d'utiliser les très bon pilotes graphique NVIDIA 3D. Je comprends et j'accepte les raisons pour lesquels NVIDIA ne souhaite pas donner les codes des pilotes. Nous travaillons TOUS pour des entreprises, et nous avons besoin d'argent, ou au moins de nourriture... Pour moi, une coexistence de l'open source et les logiciels à code source propriétaire n'est pas une mauvaise chose, quand il obéit à des règles de la licence LGPL. Je voudrais voir quelqu'un écrire un processus d'import / export CATIA pour FreeCAD et de le distribuer gratuitement ou pour de l'argent. Je n'aime pas forcer à donner plus que ce qu'il ne veut. Ce ne serait pas bon ni pour lui ni pour FreeCAD.

Néanmoins, cette décision est prise seulement pour le système de base de FreeCAD. Chaque auteur d'un module d'application peut prendre sa propre décision.

Licences utilisées

Voici les trois licences en vertu des quels FreeCAD est publié :

FreeCAD uses two different licenses, one for the application itself, and one for the documentation:

Licence publique générale limitée GNU (LGPL2+) (http://fr.wikipedia.org/wiki/Licence_publique_g%C3%A9n%C3%A9rale_limit%C3%A9e_GNU)
: Pour les bibliothèques de base telles qu'elles sont énoncées dans le .h et le .cpp dans src/App src/Gui src/Base et la plupart des modules dans src/Mod ainsi que pour l'exécutable comme indiqué dans le .h et le .cpp dans src/main. Les icônes et les autres parties graphiques sont également LGPL.

Licence publique générale GNU (GPL2+) (http://fr.wikipedia.org/wiki/Licence_publique_g%C3%A9n%C3%A9rale_GNU)
: Pour les scripts Python qui construisent les binaires comme indiqué dans les fichiers .py dans src/Tools.

Open Publication Licence
: La documentation sur http://free-cad.sourceforge.net/ ne saurait pas être décrite d'une autre façon par l'auteur.

Voir le fichier droit d'auteur FreeCAD pour debian (http://free-cad.git.sourceforge.net/git/gitweb.cgi?p=free-cad/free-cad;a=blob;f=package/debian/copyright;h=a97cf019d020edba596f2d0f614c9b09ce546b0f;hb=HEAD) (en anglais) pour plus de détails sur les licences utilisées dans FreeCAD.

Effet des licences

Les utilisateurs privés

Les utilisateurs particuliers peuvent utiliser FreeCAD gratuitement et peuvent en faire tout ce qu'ils veulent...

Les utilisateurs professionnels

Ils peuvent utiliser FreeCAD librement, pour tout type de travail privé ou professionnel. Ils peuvent personnaliser l'application comme ils le souhaitent. Ils peuvent écrire des extensions de source ouverte ou fermée à FreeCAD. Ils sont toujours maître de leurs données, ils ne sont pas obligés de mettre à jour FreeCAD, changer leur utilisation de FreeCAD. L'utilisation de FreeCAD ne les lie à aucun type de contrat ou obligation.

Développeurs open source

Ils peuvent utiliser FreeCAD comme les bases de modules d'extension propres à des fins spéciales. Ils peuvent choisir soit la licence GPL soit la LGPL pour permettre l'utilisation de leur travail dans des logiciels propriétaires ou non.

Les développeurs professionnels

Les développeurs professionnels peuvent utiliser FreeCAD comme les bases de leurs propres modules d'extension à des fins spéciales et ne sont pas obligés de faire leurs modules open source. Ils peuvent utiliser tous les modules en LGPL. Ils sont autorisés à distribuer FreeCAD avec leur logiciel propriétaire. Ils obtiendront le soutien de(s) l'auteur(s) aussi longtemps que cela n'est pas à sens unique. Si vous voulez vendre votre module, vous avez besoin d'une licence Coin3D, sinon vous êtes obligés par cette bibliothèque de le rendre open source.

OpenCasCade License side effects (for FreeCAD version 0.13 and older)

The following is no more applicable since version 0.14, since both FreeCAD and OpenCasCade are now fully LGPL.

Up to Version 0.13 FreeCAD is delivered as GPL2+, although the source itself is under LGPL2+. Thats because of linkage of Coin3D (GPL2) and PyQt(GPL). Starting with 0.14 we will be completely GPL free. PyQt will be replaced by PySide, and

Coin3D was re-licensed under BSD. One problem, we still have to face, license-wise, the OCTPL (Open CASCADE Technology Public License) (http://www.opencascade.org/getocc/license/). Its a License mostly LGPL similar, with certain changes. On of the originators, Roman Lygin, elaborated on the License on his Blog (http://opencascade.blogspot.de/2008/12/license-to-kill-license-to-use.html). The home-brew OCTPL license leads to all kind of side effects for FreeCAD, which where widely discussed on different forums and mailing lists, e.g. on OpenCasCade forum itself (http://www.opencascade.org/org/forum/thread_15859/?forum=3). I will link here some articles for the biggest problems.

GPL2/GPL3/OCTLP incompatibility

We first discovered the problem by a discussion on the FSF (http://www.fsf.org/) high priority project discussion list (https://groups.google.com/forum/#!topic/polignu/XRergtwsm80). It was about a library we look at, which was licensed with GPL3. Since we linked back then with Coin3D, with GPL2 only, we was not able to adopt that lib. Also the OCTPL is considered GPL incompatible (http://www.opencascade.org/occt/faq/). This Libre Graphics World article "LibreDWG drama: the end or the new beginning?" (http://libregraphicsworld.org/blog/entry/libredwg-drama-the-end-or-the-new-beginning) shows up the drama of LibreDWG project not acceptably in FreeCAD or LibreCAD.

Debian

The incompatibility of the OCTPL was discussed on the debian legal list (http://lists.debian.org/debian-legal/2009/10/msg00000.html) and lead to a bug report on the FreeCAD package (http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=617613) which prevent (ignor-tag) the transition from debian-testing to the main distribution. But its also mentioned thats a FreeCAD, which is free of GPL code and libs, would be acceptably. With a re-licensed Coin3D V4 and a substituted PyQt we will hopefully reach GPL free with the 0.14 release.

Fedora/RedHat non-free

In the Fedora project OpenCasCade is listed "non-free". This means basically it won't make it into Fedora or RedHat. This means also FreeCAD won't make it into Fedora/RedHat until OCC is changing its license. Here the links to the license evaluation:

- Discussion on the Fedora-legal-list (http://lists.fedoraproject.org/pipermail/legal/2011-September/001713.html)
- License review entry in the RedHat bug tracker (https://bugzilla.redhat.com/show_bug.cgi?id=458974#c10)

The main problem they have AFIK is that the OCC license demand non discriminatory support fees if you want to do paid support. It has nothing to do with "free" or OpenSource, its all about RedHat's business model!

< précédent: Dialog creation Index suivant: Tracker >

L'adresse de notre bug tracker est la suivante :

http://www.freecadweb.org/tracker

There you can report bugs, submit feature requests, patches, or request to merge your branch if you developed something using git. The tracker is divided into modules, so please be specific and file your request in the appropriate subsection. In cas of doubt, leave it in the "FreeCAD" section.

Signaler les bugs

Si vous pensez que vous pourriez avoir trouvé un bogue (dysfonctionnement ou erreur), vous êtes invité de le signaler.

Mais, avant de rapporter un bug, s'il vous plaît vérifiez les éléments suivants :

- Assurez-vous que votre bug est vraiment un bug, qu'il devrait faire quelque chose, mais il ne fonctionne pas.
- Si vous n'êtes pas sûr, n'hésitez pas à expliquer votre problème sur le forum (http://forum.freecadweb.org/) et demandez ce qu'il faut faire.
- Avant de soumettre quoi que ce soit, lisez les questions fréquemment posées (en), effectuez une recherche sur le forum (http://forum.freecadweb.org/), et assurez-vous que le même bug n'a pas déjà été signalé auparavant, en faisant une recherche sur bug tracker (http://www.freecadweb.org /tracker/main_page.php) de FreeCAD.
- Décrivez aussi clairement que possible le problème, et comment il peut être reproduit. Si nous ne pouvons pas vérifier le bug, nous ne pourrons pas être en mesure de le réparer.
- Inscrivez les informations suivantes : Votre système d'exploitation, sa version, s'il est de 32 ou 64 bits, et, la version de FreeCAD vous utilisez.
- S'il vous plaît déposer un rapport distinct pour chaque bug.
- Si vous êtes sur un système Linux, et que votre bug provoque un plantage dans FreeCAD, vous pouvez essayer de tracer le débogage :

- à partir d'un terminal exécuter **gdb FreeCAD** (en supposant que le paquet **gdb** soit installé), puis, à l'intérieur de **gdb** faire **run**.
- ensuite exécuter **FreeCAD**.
- Après que l'accident se soit reproduit, tapez **bt** , pour obtenir le **backtrace** complet.
- Inclure le **backtrace** dans votre rapport de bogue.

Demande de fonctionnalités

Si vous désirez une fonctionnalité particulière, qui n'est pas encore implémentée dans FreeCAD, ce n'est pas un bug, mais une demande de fonctionnalité.

Vous pouvez également soumettre une proposition sur **mantis bug tracker (http://www.mantisbt.org/)** même, (envoyez-la comme **demande de fonctionnalité** au lieu d'un bug), mais gardez bien à l'esprit, qu'il n'y a aucune garantie que votre souhait soit exaucé.

Soumettre un correctif (patch)

Dans le cas, où vous avez programmé une correction d'un bug (**patch**), une extension ou autre chose qui peut être d'utilité publique dans FreeCAD, créer un **patch** à l'aide de l'outil **Subversion diff tool** et de le soumettre sur **mantis bug tracker (http://www.mantisbt.org/) et envoyez-le comme patch**.

Requesting merge

Si vous avez créé une branche git contenant les modifications que vous aimeriez voir fusionné dans le code FreeCAD, vous pouvez y demander que votre branche soit examinée et fusionnée si les développeurs FreeCAD sont OK avec elle. Vous devez d'abord publier votre branche dans un dépôt git publique (github, bitbucket, sourceforge ...) et donner ensuite l'URL de votre

branche dans votre demande de fusion.

< précédent: Licence Index suivant: CompileOnWindows >

Cet article explique pas à pas comment compiler FreeCAD dans Windows.

See also Compile on Windows with Visual Studio 2013

Prérequis

Required programs

- Git (http://git-scm.com/) There are a number of alternatives such as GitCola, Tortoise Git, and others.
- CMake (http://www.cmake.org/cmake/resources/software.html) version 2.x.x or Cmake 3.x.x
- Python >2.5 (This is only required if NOT using the Libpack. The Libpack comes with a minimal Python(2.7.x) suitable for compiling and running FreeCAD)

Source Code

Using Git (Preferred)

To create a local tracking branch and download the source code you need to open a terminal(command prompt) and cd to the directory you want the source, then type:

```
git clone git://git.code.sf.net/p/free-cad/code free-cad-code
```

Compiler

On Windows, the default compiler is M$ Visual Studio, be it the Express or Full 2008, 2012, or 2013 versions. You will also need to install the Windows Platform SDK to get several required libraries (e.g. Windows.h), though they may not be required with M$ compilers (either full or express).

Note

Though it may be possible to use Cygwin or MinGW gcc it's not tested or ported so far.

Third Party Libraries

You will need all of the Third Party Libraries to successfully compile FreeCAD. If you use the M$ compilers it is recommended to install a FreeCAD LibPack (http://sourceforge.net/projects/free-cad/files/FreeCAD%20LibPack/), which provides all of the required libraries to build FreeCAD in Windows. You will need the Libpack for your architecture and compiler. FreeCAD currently supplies Libpack Version11 for x32 and x64, for VS9 2008, VS11 2012, and VS12 2013.

Optional programs

- NSIS (http://sourceforge.net/projects/nsis/) Windows installer (note: formerly, WiX (http://wixtoolset.org/) installer was used - now under transition to NSIS) - if you want to make msi installer

System Path Configuration

Inside your system path be sure to set the correct paths to the following programs:

- git (not tortoiseGit, but git.exe) This is necessary for Cmake to properly update the "About FreeCAD" information in the version.h file which allows FreeCAD to report the proper version in About FreeCAD from the help menu.
- Optionally you can include the Libpack in your system path. This is useful if you plan to build multiple configurations/versions of FreeCAD, you will need to copy less files as explained later in the build process.

To add to your system path:

- Start menu -> Right click on Computer -> Properties -> Advanced system settings
- Advanced tab -> Environment Variables...
- Add the PATH/TO/GIT to the **PATH**

- It should be separated from the others with a semicolon `;`

Configuration with CMake

The switch to CMake

Warning

Since FreeCAD version 0.9 we have stopped providing .vcproj files.

Currently, FreeCAD uses the CMake build system to generate build and make files that can be used between different operating systems and compilers. If you want build former versions of FreeCAD (0.8 and older) see Building older versions later in this article.

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

CMake

The first step to build FreeCAD with CMake is to configure the environment. There are two ways to do it:

- Using the LibPack
- Installing all the needed libraries and let CMake find them

The following process will assume you are using the LipPack. The second option may be discussed in Options for the Build Process.

Configure CMake using GUI

- Open the CMake GUI

- Specify the source folder
- Specify the build folder
- Click **Configure**
- Specify the generator according to the IDE that you'll use.

This will begin configuration and should fail because the location of **FREECAD_LIBPACK_DIR** is unset.

- Expand the **FREECAD** category and set **FREECAD_LIBPACK_DIR** to the correct location
- Check **FREECAD_USE_EXTERNAL_PIVY**
- Optionally Check **FREECAD_USE_FREETYPE** this is required to use the Draft WB's Shape String functionality
- Click **Configure** again
- There should be no errors
- Click **Generate**
- Close CMake
- Copy **libpack\bin** folder into the new build folder CMake created

Options for the Build Process

The CMake build system gives us a lot more flexibility over the build process. That means we can switch on and off some features or modules. It's in a way like the Linux kernel build. You have a lot of switches to determine the build process.

Here is the description of some of these switches. They will most likely change a lot in the future because we want to increase the build flexibility a lot more.

Link table

Variable name	Description	Default
FREECAD_LIBPACK_USE	Switch the usage of the FreeCAD LibPack on or off	On Win32 on, otherwise off
FREECAD_LIBPACK_DIR	Directory where the LibPack is	FreeCAD SOURCE dir
FREECAD_BUILD_GUI	Build FreeCAD with all Gui related modules	ON
FREECAD_BUILD_CAM	Build the CAM module, experimental!	OFF
FREECAD_BUILD_INSTALLER	Create the project files for the Windows installer.	OFF
FREECAD_BUILD_DOXYGEN_DOCU	Create the project files for source code documentation.	OFF
FREECAD_MAINTAINERS_BUILD	Switch on stuff needed only when you do a Release build.	OFF

If you are building with Qt Creator, jump to Building with Qt Creator, otherwise proceed to Building with Visual Studio 9 2008.

Building FreeCAD

Depending on your current setup, the process for building FreeCAD will be slightly different. This is due to the differences

in available software and software versions for each operating system.

The following procedure will work for compiling on Windows Vista/7/8, for XP an alternate VS tool set is required for VS 2012 and 2013, which has not been tested successfully with the current Libpacks. To target XP(both x32 and x64) it is recommended to use VS2008 and Libpack FreeCADLibs_11.0_x86_VC9.7z

Building with Visual Studio 12 2013 [afficher]

Building with Visual Studio 9 2008 [afficher]

Building with Qt Creator [afficher]

Command line build [afficher]

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 (http://sourceforge.net/project/showfiles.php?group_id=49159) on sourceforge.

You need to set the following environment variables:

FREECADLIB = "D:\Wherever\LIBPACK"

QTDIR = "%FREECADLIB%"

Add "%FREECADLIB%\bin" and "%FREECADLIB%\dll" to the system *PATH* variable. Keep in mind that you have to replace "%FREECADLIB%" with the path name, since Windows does not

recursively replace environment variables.

Directory setup in Visual Studio

Some search path of Visual Studio need to be set. To change them, use the menu *Tools→Options→Directory*

Includes

Add the following search path to the include path search list:

- %FREECADLIB%\include
- %FREECADLIB%\include\Python
- %FREECADLIB%\include\boost
- %FREECADLIB%\include\xercesc
- %FREECADLIB%\include\OpenCascade
- %FREECADLIB%\include\OpenCV
- %FREECADLIB%\include\Coin
- %FREECADLIB%\include\SoQt
- %FREECADLIB%\include\QT
- %FREECADLIB%\include\QT\Qt3Support
- %FREECADLIB%\include\QT\QtCore
- %FREECADLIB%\include\QT\QtGui
- %FREECADLIB%\include\QT\QtNetwork
- %FREECADLIB%\include\QT\QtOpenGL
- %FREECADLIB%\include\QT\QtSvg
- %FREECADLIB%\include\QT\QtUiTools
- %FREECADLIB%\include\QT\QtXml
- %FREECADLIB%\include\Gts
- %FREECADLIB%\include\zlib

Libs

Add the following search path to the lib path search list:

- %FREECADLIB%\lib

Executables

Add the following search path to the executable path search list:

- %FREECADLIB%\bin
- TortoiseSVN binary installation directory, usually "C:\Programm 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 whant to build the source code documentation you need DoxyGen (http://www.stack.nl/~dimitri/doxygen/).

To create an intstaller package you need WIX (http://wix.sourceforge.net/).

During the compilation some Python scripts get executed. So the Python interpreter has to work properly.

For more details have also a look to *README.Linux* in your sources.

First of all you should build the Qt plugin that provides all custom

widgets of FreeCAD we need for the Qt Designer. The sources are located under

```
//src/Tools/plugins/widget//.
```

So far we don't provide a makefile -- but calling

```
qmake plugin.pro
```

creates it. Once that's done, calling *make* will create the library

```
//libFreeCAD_widgets.so//.
```

To make this library known to your *Qt Designer* you have to copy the file to

```
//$QTDIR/plugin/designer//.
```

References

Template:Reflist

< précédent: Tracker Index suivant: CompileOnUnix >

On recent linux distributions, FreeCAD is generally easy to build, since all dependencies are usually provided by the package manager. It basically involves 3 steps:

1. Getting the FreeCAD source code
2. Getting the dependencies (packages FreeCAD depends upon)
3. Compiling with "cmake . && make"

Below, you'll find detailed explanations of the whole process and particularities you might encounter. If you find anything wrong or out-of-date in the text below (Linux distributions change often), or if you use a distribution which is not listed, please help us correcting it.

Getting the source

Before you can compile FreeCAD, you need the source code. There are 3 ways to get it:

Git

The quickest and best way to get the code is to clone the read-only git repository (you need the git (http://git-scm.com/) package installed):

```
git clone git://git.code.sf.net/p/free-cad/code free-cad-code
```

This will place a copy of the latest version of the FreeCAD source code in a new directory called "free-cad-code". The first time you try connecting to the free-cad.git.sourceforge.net host, you will receive a message asking to authenticate the sourceforge SSH key, which is normally safe to accept (you can check their SSH keys on the sourceforge website if you are not sure)

Github

There is an always up to date FreeCAD repository on Github: github.com/FreeCAD/FreeCAD_sf_master (https://github.com/FreeCAD/FreeCAD_sf_master)

Source package

Alternatively you can download a source package, but they could be already quite old so it's always better to get the latest sources via git or github.

- Official FreeCAD source packages (distribution-independent): https://sourceforge.net/projects/free-cad/files/FreeCAD%20Source/

Getting the dependencies

To compile FreeCAD under Linux you have to install all libraries mentioned in Third Party Libraries first. Please note that the names and availability of the libraries will depend on your distribution. Note that if you don't use the most recent version of your distribution, some of the packages below might be missing from your repositories. In that case, look in the Older and non-conventional distributions section below.

Skip to Compile FreeCAD

Debian and Ubuntu [afficher]

Fedora [afficher]

Gentoo [afficher]

OpenSUSE [afficher]

Arch Linux [afficher]

Older and non-conventional distributions [afficher]

Below is additional help for a couple of libraries that might not be present in your distribution repositories

Eigen 3

La bibliothèque Eigen3 est maintenant requise par le module Sketcher. Sous Ubuntu, cette bibliothèque n'est disponible dans les dépôts qu'à partir d'Ubuntu 11.10. Pour les versions antérieures d'Ubuntu, vous pouvez soit la télécharger ici (http://packages.ubuntu.com/oneiric/libeigen3-dev) et l'installer manuellement, ou ajouter le dépôt FreeCAD Daily Builds PPA (https://launchpad.net/~freecad-maintainers/+archive/freecad-daily) à vos sources de logiciels avant de l'installer par l'un des moyens listés ci-dessous.

OpenCASCADE community edition (OCE)

Un fork tiré d'OpenCasCade, OpenCASCADE Community edition (http://github.com/tpaviot/oce) est beaucoup plus facile à compiler. FreeCAD peut utiliser l'une ou l'autre des versions installées sur votre système, soit la version « officielle » ou la community edition. Le site Web du projet OCE contient des instructions de compilation détaillées.

OpenCASCADE official version

Note: You are advised to use the OpenCasCade community edition above, which is easier to build, but this one works too. Not all Linux distributions have an official OpenCASCADE package in their repositories. You have to check for yourself if one is available for your distribution. At least from Debian Lenny and Ubuntu Intrepid an official .deb package is provided. For older Debian or Ubuntu releases you may get unofficial packages from here (http://lyre.mit.edu/~powell/opencascade). To build your own private .deb packages follow these steps:

```
wget http://lyre.mit.edu/~powell/opencascade/opencascade_6.2.0.orig.tar.gz
wget http://lyre.mit.edu/~powell/opencascade/opencascade_6.2.0-7.dsc
wget http://lyre.mit.edu/~powell/opencascade/opencascade_6.2.0-7.diff.gz

dpkg-source -x opencascade_6.2.0-7.dsc

# Install OCC build-deps
sudo apt-get install build-essential devscripts debhelper autoconf automake libtool bison libx11-dev tcl8

#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
```

En outre, vous pouvez télécharger et compiler la dernière version disponible de opencascade.org (http://www.opencascade.org):

Installez le paquet normalement, mais sachez que l'installateur est un programme java qui nécessite l'édition officielle java runtime de Sun (nom du paquet : sun-java6-jre), pas le paquet java open-source (gij) distribué avec Ubuntu. Installez-le au besoin :

```
sudo apt-get remove gij
sudo apt-get install sun-java6-jre
```

Prenez garde, si vous utilisez gij java à d'autres applications telles qu'une extension de navigateur, elles ne fonctionneront plus. Si l'installateur ne fonctionne pas, essayez :

```
java -cp path_to_file_setup.jar <-Dtemp.dir=path_to_tmp_directory> run
```

Une fois le paquet installé, allez dans le répertoire "ros" à l'intérieur du répertoire opencascade, et faites

```
./configure --with-tcl=/usr/lib/tcl8.4 --with-tk=/usr/lib/tk8.4
```

Maintenant vous pouvez compiler. Retournez au dossier ros et faites :

```
make
```

Cela prendra beaucoup de temps, peut-être plusieurs heures.

Quand c'est terminé, installez en faisant simplement

```
sudo make install
```

Les fichiers de bibliothèque seront copiés dans /usr/local/lib ce qui est normal, puisqu'ils seront trouvés automatiquement par n'importe quel programme. En outre, vous pouvez aussi faire

```
sudo checkinstall
```

Ce qui fera la même chose que make install, mais créera une entrée dans votre système de gestion de paquets afin de le désinstaller plus facilement éventuellement. Maintenant nettoyez les considérable fichiers de compilation temporaires en faisant

```
make clean
```

Erreur possible Nº 1 : Si vous utilisez OCC version 6.2, il est fort possible que le compilateur stoppera tout juste après le début de l'opération "make". Si cela survient, éditez le script "configure", retracez la déclaration CXXFLAGS="$CXXFLAGS ", et remplacez-la par CXXFLAGS="$CXXFLAGS -ffriend-injection -fpermissive". Puis recommencez l'étape configure.

Erreur possible Nº 2 : Il est possible que plusieurs modules (WOKSH, WOKLibs, TKWOKTcl, TKViewerTest et TKDraw) se complaignent qu'ils ne trouvent pas les entêtes tcl/tk. Dans ce cas, puisque l'option n'est pas offerte par le script configure, vous devrez éditer nauellement le makefile de chacun de ces modules : Allez dans adm/make et dans chacun des dossiers des modules fautifs. Éditez le Makefile, et retracez les lignes CSF_TclLibs_INCLUDES = -I/usr/include et CSF_TclTkLibs_INCLUDES = -I/usr/include et ajoutez /tcl8.4 et /tk8.4 afin qu'elles se lisent comme suit : CSF_TclLibs_INCLUDES = -I/usr/include/tcl8.4 et CSF_TclTkLibs_INCLUDES = -I/usr /include/tk8.4

SoQt

La bibliothèque SoQt doit être compilée par rapport à Qt4, ce qui est le cas de la plupart des distributions récentes. Mais lors de l'écriture de cet article, il n'y avait des paquets SoQt4 disponibles que pour Debian, mais pas pour toutes les versions d'Ubuntu. Pour compiler les paquets, suivez les étapes suivantes :

```
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.d
```

Si votre système est en 64 bits, vous devrez probablement changer i386 par amd64.

Pivy

Pivy n'est pas nécessaire pour compiler FreeCAD ou l'exécuter, mais il est requis par le module 2D Drafting qui ne fonctionnera pas autrement. Si vous ne comptez pas utiliser ce module, vous n'avez pas besoin de pivy. Au moment d'écrire ces lignes, Pivy est très jeune et ne se trouve possiblement pas encore dans les dépôts de votre distribution. Si vous ne trouvez pas Pivy dans les dépôts de paquets de votre distribution, vous pouvez prendre des paquets debian/ubuntu sur la page de téléchargement de FreeCAD :

http://sourceforge.net/projects/free-cad/files/FreeCAD%20Linux/
or compile pivy yourself:

Pivy compilation instructions

Compiler FreeCAD

Utiliser cMake

cMake est un nouveau système de compilation dont l'avantage est d'être commun à plusieurs systèmes d'exploitation (Linux, Windows, MacOSX, etc). FreeCAD utilise désormais cMake comme système de compilation principal. La compilation avec cMake est généralement très simple et se déroule en deux étapes. À la première étape, cMake vérifie que tous les programmes et bibliothèques nécessaires sont présents sur votre système, et configure tout ce qui est nécessaire pour la compilation subséquente. Quelques alternatives vous sont détaillées ci-dessous, mais FreeCAD est livré avec des options par défaut sensées. La seconde étape est la compilation proprement

dite, qui produit l'exécutable FreeCAD.

Puisque FreeCAD est une application lourde, la compilation peut prendre un certain temps (environ 10 minutes sur un PC rapide, 30 minutes sur un PC lent).

In-source building

FreeCAD can be built in-source, which means that all the files resulting from the compilation stay in the same folder as the source code. This is fine if you are just looking at FreeCAD, and want to be able to remove it easily by just deleting that folder. But in case you are planning to compile it often, you are advised to make an out-of-source build, which offers many more advantages. The following commands will compile freecad:

```
$ cd freecad (the folder where you cloned the freecad source)
```

If you want to use your system's copy of Pivy, which you most commonly will, then set the compiler flag to use the correct pivy (via FREECAD_USE_EXTERNAL_PIVY=1). Also, set the build type to Debug if you want a debug build or Release if not. A Release build will run much faster than a Debug build. Sketcher becomes very slow with complex sketches if your FreeCAD is a Debug build. (NOTE: the "." and space after the cmake flags are CRITICAL!):

For a Debug build

```
$ cmake -DFREECAD_USE_EXTERNAL_PIVY=1 -DCMAKE_BUILD_TYPE=Debug .
$ make
```

Or for a Release build

```
$ cmake -DFREECAD_USE_EXTERNAL_PIVY=1 -DCMAKE_BUILD_TYPE=Release .
$ make
```

Your FreeCAD executable will then reside in the "bin" folder, and you can launch it with:

```
$ ./bin/FreeCAD
```

Compilation hors-source

Si vous comptez suivre l'évolution rapide de FreeCAD, il est beaucoup plus pratique de le compiler dans un dossier séparé de la source. Chaque fois que vous mettez à jour le code source, cMake distinguera intelligemment quels fichiers ont changé, et ne compilera que ce qui est requis. Les compilation hors-source sont particulièrement pratiques avec le système Git, puisque vous pouvez facilement essayer d'autres branches sans embrouiller le système de compilation. Pour compiler hors-source, créez un dossier de compilation distinct du dossier source freecad, et depuis le dossier de compilation, pointez cMake vers le dossier source :

```
mkdir freecad-build
cd freecad-build
cmake ../freecad (or whatever the path is to your FreeCAD source folder)
make
```

Votre exécutable résidera dans le dossier "bin".

Options de configuration

Il existe un certain nombre de modules expérimentaux ou inachevés que vous pourriez vouloir compiler afin de travailler sur ceux-ci. Pour ce faire, vous devez régler les options appropriées lors de l'étape de configuration. Faites-le soit en ligne de commande, en passant les options -D <var>:<type>= <value> à cMake ou en utilisant une des interfaces graphiques disponibles pour cMake (par ex. pour Debian, les paquets cmake-qt-gui ou cmake-curses-gui).

À titre d'exemple, pour configurer en ligne de commande la compilation du module Assembly, faites :

```
cmake -D FREECAD_BUILD_ASSEMBLY:BOOL=ON ''path-to-freecad-root''
```

Les options possibles sont listées dans le fichier CmakeLists.txt

situé à la racine du dossier source FreeCAD.

Greffon Qt designer

Si vous voulez faire du développement Qt pour FreeCAD, vous aurez besoin du greffon Qt designer qui fournit tous les widgets personnalisés de FreeCAD. Allez dans

```
freecad/src/Tools/plugins/widget
```

Pour l'instant nous ne fournissons pas de makefile -- mais appeler

```
qmake plugin.pro
```

le génère. Une fois que c'est fait,

```
make
```

créera la bibliothèque libFreeCAD_widgets.so. Pour faire en sorte que cette bibliothèque soit reconnue par Qt Designer, vous devez copier le fichier vers $QTDIR/plugin/designer

Doxygen

Si vous vous sentez assez audacieux pour vous plonger dans le code, vous pourriez tirer avantage à construire et consulter la documentation source de FreeCAD générée par Doxygen.

Construire un paquet Debian

Si vous envisagez de construire un paquet Debian voici les sources que vous devez installer en premier :

```
dh-make
devscripts

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

Pour construire un paquet ouvrez une console, puis il suffit d'aller

dans le répertoire FreeCAD et l'appeler

```
debuild
```

Once the package is built, you can use lintian to check if the package contains errors

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

Dépannage

Note sur les systèmes 64 bits

Pour la compilation de FreeCAD pour 64 bits, il y a un problème connu avec le paquet OpenCASCADE 64 bits. Afin que FreeCAD s'exécute correctement, vous pourriez devoir exécuter le script ./configure avec le réglage additionnel define _OCC64 :

```
./configure CXXFLAGS="-D_OCC64"
```

Sous les systèmes basés sur Debian, cette solution n'est pas requise avec l'utilisation du paquet précompilé OpenCASCADE, puisque celui-ci est déja compilé avec ce réglage. Maintenant il ne reste plus qu'à compiler FreeCAD tel que décrit ci-dessus.

Fedora 13

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/li
```

- 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
```

Automatic build scripts

Here is all what you need for a complete build of FreeCAD. It's a one-script-approach and works on a fresh installed distro. The commands will ask for root password (for installation of packages) and sometime to acknowledge a fingerprint for an external repository server or https-subversion repository. These scripts should run on 32 and 64 bit versions. They are written for different versions, but are also likely to run on a later version with or without major changes.

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

Ubuntu 13.x [afficher]

Ubuntu 14.x [afficher]

OpenSUSE 12.2 [afficher]

Debian Squeeze [afficher]

Fedora 21 [afficher]

Updating the source code

FreeCAD development happens fast, everyday or so there are bug fixes or new features. The cmake systems allows you to intelligently update the source code, and only recompile what has changed, making subsequent compilations very fast. Updating the source code with git or subversion is very easy:

```
#Replace with the location where you cloned the source code the first time
cd freecad
#If you are using git
git pull
```

Move into the appropriate build directory and run cmake again (as cmake updates the version number data for the Help menu, ...about FreeCAD), however you do not need to add the path to source code after "cmake", just a space and a dot:

```
#Replace with the location of the build directory
cd ../freecad-build
cmake .
make
```

< précédent: CompileOnWindows Index suivant: CompileOnMac >

This page explains how to compile the latest FreeCAD source code on Mac OS X.

Prerequisites

First of all, you will need to install the following software.

Xcode Development Tools

Unless you want to use the Xcode IDE for FreeCAD development, you will only need to install the Command Line Tools. To do this on 10.9 and later, open Terminal, run the following command, and then click Install in the dialog that comes up.

```
xcode-select --install
```

For other versions of OS X, you can get the package from the Apple developer downloads page (https://developer.apple.com/downloads/index.action?q=xcode) (sign in with the same Apple ID you use for other Apple services). Specifically, you will need to download Development Tools 3.2 for OS X 10.6, and Command Line Tools 4.8 for OS X 10.8.

Package Manager

You will want to use a package manager to install prerequisite software, this page gives instructions for two of the common package managers in use for OS X: Homebrew (http://brew.sh/) and MacPorts (https://www.macports.org/). It's easiest to pick one package manager for your system, and not have multiple package managers installed concurrently.

Homebrew

To install Homebrew, enter the following in Terminal:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

MacPorts

To install MacPorts, follow the instructions from their website (https://www.macports.org/install.php)

CMake

FreeCAD uses CMake (http://www.cmake.org/) to build the source. Homebrew and MacPorts can install the command line version of CMake, or if you prefer using a GUI application, install the latest version from http://www.cmake.org/download.

For the command line version of CMake, from a terminal use either Homebrew:

```
brew install cmake
```

or MacPorts:

```
sudo port install cmake
```

Installing the Dependencies

All of the needed libraries can be installed using either Homebrew or MacPorts.

Homebrew Dependencies

```
brew tap homebrew/science
brew tap sanelson/freecad
brew install boost eigen freetype oce python qt pyside pyside-tools xerces-c
brew install --without-framework --without-soqt sanelson/freecad/coin
brew install --HEAD pivy
```

MacPorts Dependencies

```
sudo port install boost eigen3 freetype oce py27-pyside-tools xercesc Coin
```

Getting the source

In this guide, the source and build folders are created in **/Users /username/FreeCAD**, but you can of course use whatever folder you want.

```
mkdir ~/FreeCAD
cd ~/FreeCAD
```

To get the FreeCAD source code, run:

```
git clone git://git.code.sf.net/p/free-cad/code FreeCAD-git
```

Alternatively, you can use the github mirror: https://github.com /FreeCAD/FreeCAD_sf_master.git

Building FreeCAD

First, create a new folder for the build:

```
mkdir ~/FreeCAD/build
```

Now you will need to run CMake to generate the build files. Several options will need to be given to CMake, which can be accomplished either with the CMake GUI application, or via the command line.

CMake Options

These instructions are valid for FreeCAD from 25 March 2015, previously several options needed to be manually specified, see the history for this page.

Name	Value	Notes
BUILD_ROBOT	0 (unchecked)	As of 12/19/2014, the robot module fails to build using newer versions of clang (OS X 10.9 and later)
CMAKE_BUILD_TYPE	Release	
FREECAD_USE_EXTERNAL_PIVY	1 (checked)	Homebrew only
FREETYPE_INCLUDE_DIR_freetype2	/usr/local /include /freetype2 for Homebrew, /opt/local /include /freetype2 for MacPorts	Only CMake version older than 3.1.0

CMake GUI

Open the CMake app, and fill in the source and build folder fields. In this case, it would be **/Users/username/FreeCAD /FreeCAD-git** for the source, and **/Users/username/FreeCAD /build** for the build folder.

Next, click the **Configure** button to populate the list of configuration options. This will display a dialog asking you to specify what generator to use. Leave it at the default **Unix Makefiles.** Configuring will fail the first time because there are some options that need to be changed. Note: You will need to

check the **Advanced** checkbox to get all of the options.

Set options from the table above, then click **Configure** again and then **Generate**.

CMake command line

Open a terminal, cd in to the build directory that was created above. Run cmake with options from the table above, following the formula -D(Name)="(Value)", and the path to your FreeCAD source directory as the final argument.

```
$cd ~/FreeCAD/build
$cmake -DBUILD_ROBOT="0" ...options continue... -DPYTHON_LIBRARY="/some/path/" ../FreeCAD-git
```

Make

Finally, from a terminal run **make** to compile FreeCAD.

```
cd ~/FreeCAD/build
make –j3
```

The -j option specifies how many make processes to run at once. One plus the number of CPU cores is usually a good number to use. However, if compiling fails for some reason, it is useful to rerun make without the -j option, so that you can see exactly where the error occurred.

If make finishes without any errors, you can now launch FreeCAD, either from Terminal with **./bin/FreeCAD**, or by double clicking the executable in Finder.

Updating

FreeCAD development happens fast; everyday or so there are bug fixes or new features. To get these changes, run:

```
cd ~/FreeCAD/FreeCAD-git
git pull
```

And then repeat the compile step above.

Troubleshooting

Fortran

"*No CMAKE_Fortran_COMPILER could be found.*" during configuration - Older versions of FreeCAD will need a fortran compiler installed. With Homebrew, do "brew install gcc" and try configuring again, for Macports, do "sudo port install gcc49" and give cmake the path to Fortran ie -DCMAKE_Fortran_COMPILER=/opt/local/bin/gfortran-mp-4.9 . Or, preferably use a more current version of FreeCAD source!

OpenGL

See OpenGL on MacOS

< précédent: CompileOnUnix suivant: Third Party Libraries >
Index

Vue d'ensemble

Ce sont des bibliothèques, qui ne sont pas modifiées dans le projet FreeCAD. elles sont inchangées, et, essentiellements utilisées comme bibliothèques de liens dynamiques (***.So (http://fr.wikipedia.org /wiki/Bibliothèque_logicielle#Unix.2C_GNU.2FLinux_et_BSD** ou ***.Dll (http://fr.wikipedia.org /wiki/Dynamic_Link_Library)**). S'il y a un changement nécessaire, ou une classe wrapper est nécessaire, le code du package, ou le code de la bibliothèque ont changés et doivent être déplacés vers le package de base de FreeCAD. Les bibliothèques utilisées sont les suivantes :

Pensez à utiliser LibPack au lieu de télécharger et d'installer toutes sorte de trucs.

Liens

Link table

Nom de la Lib	Version nécessaire	Lien pour l'obtenir
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 ou plus

License: Python 2.5 licence

Vous pouvez utiliser le source ou binaire à partir de Python (http://www.python.org/) ou utiliser alternativement **ActiveState Python** à partir de activestate (http://www.activestate.com/) s'il est difficile d'obtenir des libs de débogage à partir d'**ActiveState**.

Description

Python, est le langage de script principal, et, est utilisé dans toute l'application. Par exemple :

- Mettre en œuvre des scripts de test pour tester :
  - des pertes de mémoire.
  - d'assurer de nouvelles fonctionnalités après modifications.
  - poster, construire des contrôles.
  - des tests de contrôles de tests.
- Macros et enregistrements de macros.
- Mettre en œuvre une logique d'application, pour les paquets (packages) standards.
- La mise en œuvre des boîtes à outils complètes.
- Le chargement dynamique des paquets (packages).
- Les règles d'application pour la conception (connaissances techniques).
- Créer par exemple des groupes de travail et PDM sur Internet.
- Et ainsi de suite ...

Le chargement de packages dynamiques pour Python est utilisé, en particulier, au moment de l'exécution, pour le chargement de fonctionnalités supplémentaires, et, établit le nécessaires pour les tâches réelles. Pour voir Python de plus près : Pourquoi Python direz vous ? vous pouvez le demander ici

(http://www.python.org/). Il y a plusieurs raisons : Jusqu'à présent, dans ma vie professionnelle, j'ai utilisé les langages de script différents :

- Perl
- Tcl/Tk
- VB
- Java

Python est plus orienté OO (object-oriented), le code n'est pas plus mauvais que Perl et Tcl, pareil pour Perl et VB. Java n'est pas un langage destiné au script, et, difficile (voire impossible) à intégrer. Python, est bien documenté, facile à intégrer, et, facile à étendre. Il est également bien fait ses preuves, et, est fort prisé dans la communauté open source.

Credits

Grâce à Guido van Rossum (http://fr.wikipedia.org /wiki/Guido_van_Rossum) et beaucoup de gens, ont fait que Python ait un tel succès !

OpenCasCade

Version: 5.2 ou plus

License : OCTPL

OCC (http://www.opencascade.org/) est un noyau complet **CAD**. A l'origine, il a été développé en France par **Matra Datavision**, pour la **Strim (Styler)** et **Euclide applications quantiques**, et, plus tard fait pour l'Open Source. C'est une bibliothèque vraiment énorme, et, faire en premier lieu une application de CAO libre est possible, en fournissant certains paquets, qui seraient difficiles, ou impossibles à mettre en œuvre dans un projet Open Source :

- Un noyau géométrique complet conforme à **STEP**.
- Un modèle topologique de données et toutes les fonctions nécessaires pour travailler sur les (coupes, fusion, extrusion,

etc ...)
- Import-standard/exportation des processeurs comme STEP (http://fr.wikipedia.org/wiki/STEP-NC), IGES (http://fr.wikipedia.org /wiki/Initial_Graphics_Exchange_Specification), VRML (http://fr.wikipedia.org /wiki/Virtual_Reality_Markup_Language).
- Visionneuse 2D et 3D avec le soutien de la sélection.
- Une structure de document, et, données de projet, avec le soutien de, sauvegarde et restauration, de liaison externe des documents, de recalcul de l'historique du dessin (modélisation paramétrique) et d'un centre de chargement de nouveaux types de données, comme un module d'extension dynamique.

Pour en savoir plus sur OpenCascade jeter un coup oeil à la page OpenCascade ou sur OpenCascade (http://www.opencascade.org).

Qt

Version: 4.1.x or higher

Licence : GPL v2.0/v3.0 ou commerciale (à partir de la version 4.5 aussi sur v2.1 LPGL)

Je ne pense pas que j'ai besoin de dire beaucoup de choses sur Qt. C'est un des outils les plus souvent utilisés, dans l'interface graphique des projets Open Source. Pour moi, le point le plus important d'utiliser Qt est le **Qt Designer** et la possibilité de charger les boîtes de dialogue entières comme, une ressource (XML), et, d'intégrer des widgets spécialisés.

Dans une application CAX, l'interaction avec l'utilisateur, et, les boîtes de dialogue, sont de loin la plus grande partie du code, et, un bon concepteur de boîtes de dialogues, est très important pour ajouter facilement de nouvelles fonctionnalités à FreeCAD.

Vous trouverez de plus amples informations, et une très bonne documentation en ligne sur Qt (http://www.qtsoftware.com)

Coin3D

Version: 2.0 ou plus

License: GPL v2.0 ou Commercial

Coin (http://www.coin3d.org/) est une bibliothèque graphique 3D de haut niveau, avec une interface de programmation C++. Coin utilise une structure de données scenegraph, pour rendre des graphiques en temps réel, il est adapté à toutes sortes d'applications de visualisation scientifique, et, d'ingénierie.

Coin est portable sur un large éventail de plates-formes : tous les systèmes UNIX (http://fr.wikipedia.org/wiki/Unix) / Linux (http://fr.wikipedia.org/wiki/Linux) / BSD (http://fr.wikipedia.org/wiki/Berkeley_Software_Distribution), tous les systèmes d'exploitation Microsoft Windows, et Mac OS X.

Coin est construit sur le standard industriel OpenGL (http://fr.wikipedia.org/wiki/OpenGL) avec les bibliothèques de rendu immédiat, et, ajoute les abstractions de primitives de haut niveau, fournit une interactivité 3D, augmente considérablement la commodité et la productivité du programmeur, contient de nombreuses fonctions d'optimisations complexes, pour obtenir un rendu rapide, et, de plus est transparent pour le programmeur d'applications.

Coin est basé sur l'**API SGI Open Inventor**. Pour ceux qui ne sont pas familier avec lui, dans la communauté scientifique et d'ingénierie, Open Inventor est depuis longtemps, devenu de facto, la bibliothèque graphique standard pour la visualisation 3D et pour les logiciels de simulation visuelle. Sur une période de plus de 10 ans, il a prouvé, qu'il en vaut la peine, sa maturité contribue à son succès, en tant que fondation majeure dans des milliers d'applications d'ingénierie de grande envergure à travers le monde.

Nous allons utiliser OpenInventor en tant que visualiseur 3D dans FreeCAD parce que les visualiseurs OpenCascade (AIS et Graphics3D) ont leurs limites, à cause de grands flux de données,

et, quand il y a des rendus d'ingénierie à grande échelle. D'autres choses, comme les textures ou le rendu volumétrique ne sont pas bien pris en charge, et ainsi de suite ....

Depuis la version 2.0 Coin utilise un modèle de licence différente. Ce n'est plus LGPL (http://fr.wikipedia.org /wiki/Licence_publique_générale_limitée_GNU). Pour l'Open source, ils utilisent le GPL (http://fr.wikipedia.org /wiki/Licence_publique_générale_GNU), et, une licence commerciale pour le source fermé. Cela signifie que si vous voulez vendre votre ouvrage basé sur FreeCAD (modules d'extension), vous **devez acheter** une licence Coin !

SoQt

Version: 1.2.0 ou plus

License: GPL v2.0 ou commercial

SoQt est l'inventeur de la liaison avec la boîte à outils **Qt Gui**. Malheureusement, il n'est plus LGPL, et, nous devons donc le supprimer du code de FreeCAD, et, le lier comme une bibliothèque. Il a le même type de licence que **Coin**. Et vous devez le compiler avec votre version de **Qt**.

Xerces-C++

Version: 2.7.0 ou plus

License: Apache Software License Version 2.0

Xerces-C++ (http://xerces.apache.org/xerces-c/) est un analyseur de validation XML, écrit dans un sous-ensemble portable de C++. Avec Xerces-C++, il est facile de donner à votre application la capacité de lire et écrire des données au format XML (http://fr.wikipedia.org/wiki/Extensible_Markup_Language). Une bibliothèque partagée est prévue pour l'analyse, la génération, la manipulation et la validation des documents **XML (http://fr.wikipedia.org/wiki/Extensible_Markup_Language)**.

Xerces-C++, est fidèle à la recommandation **XML 1.0** et de nombreuses normes connexes (voir Caractéristiques ci-dessous).

L'analyseur fournit, de hautes performances, la modularité et l'évolutivité. Code source, les échantillons et documentation de l'API (http://fr.wikipedia.org/wiki/Interface_de_programmation) sont fournis avec l'analyseur. Pour la portabilité, nous avons pris soin de faire une utilisation minimale de modèles, pas de RTTI (http://fr.wikipedia.org/wiki/Run-time_type_information), et l'utilisation minimale de #ifdef.

L'analyseur est utilisé, pour sauvegarder, et, restaurer les paramètres dans FreeCAD.

Zlib

Version: 1.x.x

License: zlib Licence

zlib est conçu pour comprimer des données de toute sorte, il est libre, et légalement utilisé, il n'est pas couvert par des brevets, il compresse sans perte de données, et pour une utilisation sur pratiquement n'importe quel matériel informatique et système d'exploitation. Le format des données **zlib** est lui-même portable sur toutes les plateformes. Contrairement à la méthode de compression **LZW (http://fr.wikipedia.org/wiki/Lempel-Ziv-Welch)** utilisée sous Unix compress(1) et dans le format d'image GIF (http://fr.wikipedia.org /wiki/Graphics_Interchange_Format), la méthode de compression utilisée actuellement dans zlib, ne "gonfle" jamais les données. (LZW peut doubler ou dans les cas extrêmes, tripler la taille du fichier). L'empreinte mémoire de la librairie **zlib**, est également indépendante des données entrées et peut être, si nécessaire, réduite à un certain taux de compression.

Boost

Version: 1.33.x

License: Boost Software License - Version 1.0

Les bibliothèques Boost C++ sont une collection évaluées par des pairs, les bibliothèques, sont open source, et, étendent les fonctionnalités de C++. Les bibliothèques sont sous licence **Boost Software License**, Boost est conçu, pour être utilisé avec des projets **open source** et **fermés**. Beaucoup de programmeurs Boost sont sur le **C++ standard committee**, et plusieurs bibliothèques Boost ont été acceptées, pour leurs incorporations dans le **Technical Report 1 of C++0x**.

Les bibliothèques Boost sont en C++, et, destinées à un large éventail de programmeurs et un vaste domaine d'applications. Les bibliothèques sont conçues à des fins générales, comme pour **SmartPtr**, à des applications comme OS et FileSystem, et a des bibliothèques principalement destinées aux développeurs de bibliothèques et d'autres utilisateurs avancés en C++, comme la bibliothèque MPL (http://fr.wikipedia.org /wiki/Mozilla_Public_License).

Afin d'assurer l'efficacité et la flexibilité, Boost fait un usage intensif de modèles (templates). Boost a été une source de travail, et, de recherches approfondies dans la programmation générique, et, méta-données en C++.

Allez voir sur : boost (http://www.boost.org/) pour plus de détails.

LibPack

LibPack est un package pratique, avec toutes les bibliothèques décrites ci-dessus, en un seul paquet. Il est actuellement disponible pour la plate-forme Windows, sur la page de téléchargement ! Si vous travaillez sous Linux, vous n'avez pas besoin d'un LibPack, à la place, utilisez les dépôts (package repositories) de votre distribution Linux.

FreeCADLibs7.x Changelog

- Utilisation de QT 4.5.x et Coin 3.1.x
- Eigen ajout de template lib pour Robot

- SMESH expérimental

< précédent: CompileOnMac Index suivant: Third Party Tools >

Page d'outils

Pour chaque développement de logiciels sérieux, vous avez besoin d'outils sérieux. Voici une liste d'outils, que nous utilisons pour développer FreeCAD :

Outils indépendants de la plate-forme

Qt-Toolkit

Qt-toolkit est un outil de conception d'interfaces utilisateur, indépendamment de la plate forme utilisée. Elle est contenue dans le **LibPack** de FreeCAD, mais peut aussi être téléchargé à l'adresse Qt project (http://qt-project.org/downloads).

InkScape

Excellent programme de dessin vectoriel. Adhère à la norme SVG (http://fr.wikipedia.org/wiki/Scalable_Vector_Graphics), et, est utilisé pour dessiner les icônes et les images. Pour le télécharger, allez sur inkscape (http://inkscape.org/?lang=fr&css=css/base.css).

Doxygen

Un très bon outil, stable, il génère de la documentation à partir de fichiers sources **.h** et **.cpp** .

Gimp

Pas grand chose à dire sur le célèbre Gnu Image Manipulation Program. Outre, qu'il peut gérer les fichiers **.Xpm**, qui est un moyen très pratique pour créer les icônes dans le programme Qt-Toolkit. Le format **.XPM** est fondamentalement **C-Code**, qui peut être compilé, dans un programme comme Qt-Toolkit.

Téléchargez la dernière version de GIMP ici (http://www.gimp.org/)

Outils pour Windows

Visual Studio 8 Express

Bien que VC8 pour le développement en C++, n'est pas vraiment un pas en avant depuis VisualStudio 6 (plutôt un grand pas en arrière), soit un système de développement libre sur Windows. Pour les applications natives Win32, vous devez télécharger le **PlatformSDK de M$ (http://www.microsoft.com/en-us /download/details.aspx?id=6510)**, l'édition Express est difficile à trouver.

Mais vous pouvez essayer ce lien Visual Studio Express (http://msdn.microsoft.com/vstudio/express/visualc/default.aspx).

CamStudio

CamStudio est un outil Open Source pour créer des enregistrements vidéos d'écran (Webcasts). C'est un très bon outil, pour créer des tutoriels vidéos (avec ou sans son), en enregistrant toutes vos opérations et mouvements de souris, qui se passent sur votre écran . Une vidéo est bien moins ennuyeuse, que l'écriture d'une documentation.

Vous pouvez aller voir le site de camstudio (http://camstudio.org/) pour plus de détails.

Tortoise SVN

Il s'agit d'un très bon outil. Il rend l'utilisation de Subversion (notre système de contrôle de versions sur sf.net) en un réel plaisir. Vous pouvez penser à l'intégration de l'explorateur, de gérer facilement des révisions, de consulter les différences, de résoudre les conflits, assurer les branches, et ainsi de suite .... La boîte de dialogue en elle-même est une œuvre d'art. Elle vous donne un aperçu sur vos fichiers modifiés et vous permet de les valider ou non. Il est alors facile de rassembler les modifications

apportées aux unités logiques et de leurs donner un message clair de validation.

Vous trouverez sur tortoisesvn.tigris.org (http://tortoisesvn.tigris.org/).

StarUML

StarUML est un programme Open Source. Il a beaucoup de caractéristiques des grands, y compris l'engeniering inverse du code source C++ ....

Téléchargez le ici : staruml.sourceforge.net (http://staruml.sourceforge.net/en/)

Outils pour Linux

A venir.

< précédent: Third Party Libraries Index
suivant: Start up and Configuration >

Cette page montre, les différentes façons de lancer FreeCAD, et, ses configurations les plus importantes.

Démarrer FreeCAD en ligne de commande

FreeCAD peut être lancé normalement, en double-cliquant sur son icône qui est sur le bureau, ou, en le sélectionnant dans le menu de démarrage, mais, il peut également être lancé directement à partir de la ligne de commande. Cela vous permet de changer les options de démarrage par défaut **SOEM**.

Les options disponibles en ligne de commande

Les options en ligne de commande sont l'objet de fréquents changements, il est donc sage de vérifier les options de votre version courante en tapant :

```
FreeCAD --help
```

Les réponses disponibles, sont dans les paramètres :

```
Usage:
FreeCAD [options] File1 File2 .....
Allowed options:

Generic options:
 -v [ --version ]      print version string
 -h [ --help ]         print help message
 -c [ --console ]      start in console mode
 --response-file arg   can be specified with '@name', too
```

```
Configuration:
 -l [ --write-log ] arg    write a log file to default location(Run FreeCAD --h to see default location)
 --log-file arg            Unlike to --write-log this allows to log to an arbitrary file
 -u [ --user-cfg] arg      User config file to load/save user settings
 -s [ --system-cfg] arg    System config file to load/save system settings
 -t [ --run-test ] arg     test level
 -M [ --module-path ] arg  additional module paths
 -P [ --python-path ] arg  additional python paths
```

EX: (Windows)

```
"C:\Program Files\FreeCAD 0.14\bin\FreeCAD.exe" -M "N:\FreeCAD\Mod\Draft" -M "N:\FreeCAD\Mod\Part" -M "N
```

"Response" fichiers de configurations

Vous pouvez lire certaines options de FreeCAD à partir d'un fichier de configuration. Ce fichier doit être dans le répertoire **/bin** et doit être nommé **FreeCAD.cfg**. **Notez, que les options spécifiées en ligne de commande, remplacent le fichier de configuration !**

Certains systèmes d'exploitation ont une limite assez courte de la longueur de la chaîne, en ligne de commande. La façon courante de contourner ces limitations, est l'utilisation des fichiers de **Response**. Un fichier de **Response** n'est qu'un fichier de configuration, qui utilise la même syntaxe qu'a la ligne de commande. Si la ligne de commande spécifie un nom de fichier de **Response** à utiliser, il est chargé analysé, et s'ajoute à la ligne de commande :

```
FreeCAD @ResponseFile.txt
```

ou :

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

Options cachées

Il y a des options qui sont invisibles à l'utilisateur. Ces options sont par exemple, les paramètres X-Window analysés par le système Windows:

- **-display display**, définit l'affichage X (valeur par défaut est $DISPLAY).
- **-geometry geometry**, la géométrie fixe de la première fenêtre client qui est affichée.
- **-fn or -font font**, définit la police de l'application. La police doit être spécifié en utilisant la X logical font description.
- **-bg or -background color**, définit la couleur de fond par défaut et une palette d'applications (tons clairs et foncés sont

calculés).
- **-fg or -foreground color**, définit la couleur de premier plan par défaut.
- **-btn or -button color**, définit la couleur des boutons par défaut.
- **-name name**, définit le nom de l'application.
- **-title title**, définit le titre de l'application.
- **-visual TrueColor**, force l'application à utiliser un visuel TrueColor sur un affichage 8-bits.
- **-ncols count**, limite le nombre de couleurs allouées dans le cube de couleur sur un écran 8-bits, si l'application utilise la spécification de couleur QApplication::ManyColor. Si le nombre est 216, puis un cube 6x6x6 couleurs est utilisé (soit 6 niveaux de rouge, 6 de vert, et 6 de bleu); pour d'autres valeurs, un cube à peu près proportionnel à un cube 2x3x1 couleurs est utilisé.
- **-cmap**, provoque l'installation d'une carte de couleurs privées à l'application, sur un affichage 8-bits.

Démarrer FreeCAD sans interface utilisateur

Normalement, FreeCAD démarre en mode graphique (GUI), mais vous pouvez aussi le forcer à démarrer en mode console en tapant :

```
FreeCAD -c
```

En ligne de commande. En mode console, aucune interface utilisateur, ne sera affichée, et l'invite vous sera présenté avec un interpréteur Python.

A partir de ce prompt Python, vous avez les mêmes fonctionnalités que l'interpréteur Python qui fonctionne au sein de l'interface graphique de FreeCAD, et, un accès normal à tous les modules et plugins de FreeCAD, à l'exception du module **FreeCADGui**. Notez que les modules qui dépendent de **FreeCADGui** peuvent également être inaccessibles.

Exécuter FreeCAD comme un module Python

FreeCAD peut également être utilisé et exécuté en tant que module Python à l'intérieur d'autres applications, qui utilisent Python, ou, à partir d'un shell Python externe. Pour cela, l'application hôte Python doit **savoir où résident** vos libs FreeCAD. La meilleure façon de l'obtenir, c'est d'annexer temporairement le chemin des libs de FreeCAD à la variable **sys.path**. Le code suivant tapé à partir de n'importe quel shell Python va importer FreeCAD, et vous permettre de l'exécuter de la même manière que dans **le mode console** :

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

Une fois que FreeCAD est chargé, c'est à vous de le faire interagir avec votre application hôte de toutes les manières que vous pouvez imaginer !

Ensemble de configuration

A chaque démarrage, FreeCAD examine ses environs, ainsi que les paramètres en ligne de commande. Il construit un ensemble de configurations qui détiennent le cœur des informations d'exécution. Ces informations sont ensuite utilisées pour déterminer l’emplacement, où enregistrer les données des utilisateurs ou des fichiers journaux. Il est également très important après analyse post-mortem. Par conséquent, il est enregistré dans le fichier journal (log file).

Informations correspondantes à l'utilisateur

L'appel se fait de la manière suivants :

```
path = FreeCAD.ConfigGet("UserAppData")
```

User config entries

Config nom var	Synopsis	Exemple M$	Exemple (Linu
UserAppData	Chemin où FreeCAD met les données utilisateur de l'application.	C:\Documents and Settings\username \Application Data\FreeCAD	/home/use /.FreeCAD
UserParameter	Chemin où FreeCAD met les fichier utilisateur de l'application.	C:\Documents and Settings\username \Application Data\FreeCAD \user.cfg	/home/use /.FreeCAD /user.cfg
SystemParameter	Fichier où sont les données de l'application.	C:\Documents and Settings\username \Application Data\FreeCAD \system.cfg	/home/use /.FreeCAD /system.cf
UserHomePath	Chemin racine de l'utilisateur courant.	C:\Documents and Settings\username\My Documents	/home/use

Arguments en ligne de commande

User config entries

Config nom var	Synopsis	Exemple
LoggingFile	1 si l'enregistrement est activé	1

LoggingFileName	Nom où est placé le fichier journal	C:\Documents and Settings\username\Application Data\FreeCAD\FreeCAD.log
RunMode	Cela indique comment la boucle principale travaillera. **"Script"** signifie que le script donné est appelé puis quitté. **"Cmd"** est destiné à l'interpréteur en ligne de commande. **"Internal"** exécute un script interne. **"Gui"** entre dans la boucle d'évènement Gui. **"Module"** charge un module Python donné.	"Cmd"
FileName	Dépend du RunMode	
ScriptFileName	Dépend du RunMode	
Verbose	Niveau de commentaire de FreeCAD	"" or "strict"
OpenFileCount	Donne le nombre de dossiers ouverts par les arguments en ligne de commande	"12"
AdditionalModulePaths	Contient les chemins, des modules supplémentaires	"extraModules/"

	donnés dans la ligne de commande	

Systèmes liés

L'appel se fait de la manière suivants :

```
path = FreeCAD.ConfigGet("AppHomePath")
```

User config entries

Config var name	Synopsis	Exemple M$	Exemple Posix (Linux)
AppHomePath	Chemin où est installé FreeCAD	c:/Progam Files/FreeCAD_0.7	/user/local /FreeCAD_0.7
PythonSearchPath	Donne une liste de chemins que les modules Python recherchent. S'effectue au démarrage, et peut changer en cours d'exécution		

Certaines bibliothèques, ont besoin d'appeler les variables d'environnement système. Parfois, il y a des problèmes avec une installation de FreeCAD, c'est parce que certaines variables d'environnements sont absentes ou mal réglées. Par conséquent, certaines variables importantes se reproduisent dans la configuration et enregistrées dans le fichier journal (log file).

Variables d’environnement relatifs à Python :

- PYTHONPATH
- PYTHONHOME
- TCL_LIBRARY
- TCLLIBPATH

Variables d’environnement relatifs à OpenCascade :

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

Variables d’environnement relatifs au Système :

- PATH

Construire des informations connexes

Le tableau ci-dessous montre les informations générées par la version disponible. La plupart viennent du dépôt de Subversion. Cette astuce est nécessaire pour reconstruire exactement une version !

User config entries

Config var name	Synopsis	Exemple
BuildVersionMajor	Numéro de version majeure de la construction. Définie dans src/Build	0

	/Version.h.in	
BuildVersionMinor	Numéro de version mineure de la construction. Définie dans src/Build /Version.h.in	7
BuildRevision	Nombre SVN révision du référentiel du src dans la construction. Généré par SVN	356
BuildRevisionRange	Gamme de changements différentes	123-356
BuildRepositoryURL	Repository URL	https://free-cad.svn.sourceforge.net /svnroot/free-cad/trunk/src
BuildRevisionDate	Date de la révision susmentionnée ci-dessus	2007/02/03 22:21:18
BuildScrClean	Indicates if the source was changed ager checkout	Src modified
BuildScrMixed		Src not mixed

Image de marque liée

Ces entrées de configuration sont liées au mécanisme de l'image de marque de FreeCAD. Voir Branding pour plus de

renseignements.

User config entries

Config nom var	Synopsis	Exemple
ExeName	Nom du fichier exécutable de compilation. Ce nom peut être différent de FreeCAD si un **main.cpp** différent est utilisé.	FreeCAD.exe
ExeVersion	La version présente au moment de la compilation	V0.7
AppIcon	L'icône qui est utilisé pour l'exécutable, affichée dans application MainWindow	"FCIcon"
ConsoleBanner	Bannière qui est invité en mode console	
SplashPicture	Nom de l'icône utilisée pour l'écran de démarrage	"FreeCADSplasher"
SplashAlignment	Alignement du texte dans la boîte de dialogue Splash	Left"
SplashTextColor	Couleur du texte splasher	"#000000"
StartWorkbench	Nom du Workbech qui commence automatiquement après le démarrage	"Part design"
HiddenDockWindow	Liste des dockwindows (séparés par un point-virgule) qui seront	"Property editor"

	désactivés	

< précédent: Third Party Tools suivant: FreeCAD Build Tool >
Index

L'outil **de construction de FreeCAD** ou **fcbt** est un script python situé à :

```
trunc/src/Tools/fcbt.py
```

Il peut être utilisé, pour simplifier certaines tâches fréquemment utilisées dans la construction (compilation), la distribution, et, l'extension de **FreeCAD**.

Utilisation

Quand Python (http://fr.wikipedia.org/wiki/Python_(langage)) est correctement installé, **fcbt** peut être invoqué par la commande :

```
python fbct.py
```

Il affiche un menu, où vous pouvez sélectionner la tâche, que vous souhaitez utiliser pour :

```
FreeCAD Build Tool
 Usage:
    fcbt <command name> [command parameter]
 possible commands are:
  - DistSrc         (DS)   Build a source Distr. of the current source tree
  - DistBin         (DB)   Build a binary Distr. of the current source tree
  - DistSetup       (DI)   Build a Setup Distr. of the current source tree
  - DistSetup       (DUI)  Build a User Setup Distr. of the current source tree
  - DistAll         (DA)   Run all three above modules
  - NextBuildNumber (NBN)  Increase the Build Number of this Version
  - CreateModule    (CM)   Insert a new FreeCAD Module in the module directory

 For help on the modules type:
   fcbt <command name> ?
```

À l'invite de commande, entrez la commande abrégée que vous voulez appeler. Par exemple, tapez **«CM»** pour la création d'un module.

DistSrc

La commande "DS" Crée le source de la distribution de l'arbre source de courant.

DistBin

La commande "DB" Créer une distribution binaire de l'arbre source de courant.

DistSetup

La commande "DI" crée une distribution d'installation de l'arbre source de courant.

DistSetup

La commande "DUI" crée une distribution de configuration utilisateur de l'arbre source de courant.

DistAll

La commande "DA" exécute la séquence **"DS", "DB"** et **"DI"**.

NextBuildNumber

La commande "NBN" incrémente le numéro de compilation pour créer une nouvelle version de FreeCAD.

CreateModule

La commande "CM" crée un nouveau module de l'application.

< précédent: Start up and Configuration Index
suivant: Module Creation >

Ajouter de nouveaux **modules** et **boîtes à outils** dans FreeCAD est très facile. Nous appelons **module**, toute extension de FreeCAD, tandis qu'un **plan de travail** (workbench) est une configuration spéciale **GUI (http://fr.wikipedia.org /wiki/Interface_graphique)**, habituellement, les groupes de **barres d'outils** et de **menus**. Vous créez un nouveau module qui contient son propre plan de travail (sa barre d'outils et ses commandes).

Les modules peuvent être programmés en C++ (http://fr.wikipedia.org/wiki/C%2B%2B) ou en Python (http://fr.wikipedia.org/wiki/Python_(langage)), ou un mélange des deux, mais les fichiers de module d'initialisation, doivent être en Python. La mise en place d'un nouveau module, avec les fichiers d'initialisation est facile, et, peut être effectuée, soit manuellement, soit avec l'outil **build de FreeCAD**.

Utilisation des outils de FreeCAD

La création d'un nouveau module dans FreeCAD est assez simple. Dans l'arborescence de développement de FreeCAD, il existe l'outil FreeCAD Build Tool (fcbt) qui, fait les choses les plus importantes pour vous.
Il s'agit d'un script Python (http://fr.wikipedia.org /wiki/Python_(langage)) situé à :

```
trunk/src/Tools/fcbt.py
```

Lorsque votre interpréteur **Python (http://fr.wikipedia.org /wiki/Python_(langage))** est correctement installé, vous pouvez exécuter le script en ligne de commande avec :

```
python fcbt.py
```

Le menu suivant s'afficher :

```
FreeCAD Build Tool
Usage:
   fcbt <command name> [command parameter]
```

```
possible commands are:
  - DistSrc         (DS)   Build a source Distr. of the current source tree
  - DistBin         (DB)   Build a binary Distr. of the current source tree
  - DistSetup       (DI)   Build a Setup Distr. of the current source tree
  - DistSetup       (DUI)  Build a User Setup Distr. of the current source tree
  - DistAll         (DA)   Run all three above modules
  - BuildDoc        (BD)   Create the documentation (source docs)
  - NextBuildNumber (NBN)  Increase the Build Number of this Version
  - CreateModule    (CM)   Insert a new FreeCAD Module in the module directory

For help on the modules type:
   fcbt <command name> ?
```

À l'invite de comande, entrez **CM** pour commencer la création d'un module :

```
Insert command: ''CM''
```

Vous êtes maintenant invité à spécifier un nom pour votre nouveau module.
Appelons le **TestMod** par exemple :

```
Please enter a name for your application: ''TestMod''
```

Après avoir validé, **fcbt** commence à copier, tous les fichiers nécessaires pour votre module dans un nouveau dossier, à :

```
trunk/src/Mod/TestMod/
```

Puis, tous les fichiers sont modifiés avec votre nouveau nom de module. La seule chose que vous devez faire maintenant, est d'ajouter les deux nouveaux projets, **"appTestMod"** et **"appTestModGui"**, à votre espace de travail (sous Windows) ou à vos objectifs Makefile (unix). C'est tout !

Mise en place d'un nouveau module manuellement

Vous avez besoin de deux choses, pour créer un nouveau module :

- Un nouveau dossier dans le dossier **Mod** de FreeCAD (soit dans **Installationd_Path/FreeCAD/Mod** ou dans **UserPath/.FreeCAD/Mod**). Vous pouvez le nommer comme

vous le souhaitez.
- Dans ce dossier, il y a un fichier **InitGui.py**. Ce fichier sera automatiquement exécuté au démarrage de FreeCAD (par ex, mettre un print("Bonjour tout le monde") à l'intérieur)

En outre, vous pouvez également ajouter un fichier **Init.py**. La différence est, que le fichier **InitGui.py** n'est chargé que lorsque FreeCAD fonctionne en mode graphique (Init**GUI**), et, le fichier **Init.py** est **toujours** chargé. Mais si nous faisons un plan de travail (workbench), nous allons le mettre en **InitGui.py**, parce les outils, sont utilisés uniquement en mode **GUI**, bien sûr.

Création de nouveaux outils

Une des premières choses que vous voudrez faire, est de définir un plan de travail dans le fichier **InitGui.py**.
Voici un petit code que vous pouvez utiliser :

```
class MyWorkbench ( Workbench ):
       "My workbench object"
       Icon = """
                        /* XPM */
                        static const char *test_icon[]={
                        "16 16 2 1",
                        "a c #000000",
                        ". c None",
                        "................",
                        "................",
                        "..############..",
                        "..############..",
                        "..############..",
                        "......####......",
                        "......####......",
                        "......####......",
                        "......####......",
                        "......####......",
                        "......####......",
                        "......####......",
                        "......####......",
                        "......####......",
                        "................",
                        "................"};
                        """
       MenuText = "My Workbench"
       ToolTip = "This is my extraordinary workbench"

        def GetClassName(self):
               return "Gui::PythonWorkbench"

       def Initialize(self):
               import myModule1, myModule2
               self.appendToolbar("My Tools", ["MyCommand1","MyCommand2"])
               self.appendMenu("My Tools", ["MyCommand1","MyCommand2"])
               Log ("Loading MyModule... done\n")

       def Activated(self):
               # do something here if needed...
```

```
                Msg ("MyWorkbench.Activated()\n")

        def Deactivated(self):
                # do something here if needed...
                Msg ("MyWorkbench.Deactivated()\n")

FreeCADGui.addWorkbench(MyWorkbench)
```

L'atelier (boîte à outils) doit disposer de toutes ces définissions (attributs) :

- **Icon** L'attribut **Icon** est une image **XPM (http://fr.wikipedia.org/wiki/X_PixMap)** (La plupart des logiciels tel que GIMP (http://www.gimp.org/) permet de convertir une image en format xpm, qui, est un simple fichier texte. Vous pouvez ensuite coller le contenu ici).
- **MenuText** est le nom établi tel qu'il apparaîtra dans la liste établis (boîte à outils).
- **Tooltip** (Info-bulle) s'affiche lorsque vous le survolez avec la souris.
- **Initialize()** est exécuté au chargement de FreeCAD, et doit créer tous les menus, et, barres d'outils que le plan de travail (workbench) va utiliser. Si vous faites votre module en C++, vous pouvez aussi définir vos menus et barres d'outils à l'intérieur du module C++, **et pas** dans le fichier InitGui.py. L'important est, qu'il soit créé maintenant, et pas lorsque le module est activé.
- **Activated()** est exécuté, lorsque l'utilisateur bascule sur votre plan de travail (module).
- **Deactivated()** est exécuté, lorsque l'utilisateur bascule de vôtre atelier (module), à un autre atelier (module) ou, quitte FreeCAD

Creation de commandes FreeCAD en Python

Habituellement, vous définissez tous vos outils (appelés commandes dans FreeCAD), dans un autre module, puis importez ce module, avant de créer les barres d'outils et de menus.
Il s'agit ici d'un code minimum, que vous pouvez utiliser pour définir une commande :

```
import FreeCAD,FreeCADGui

class MyTool:
	"My tool object"

	def GetResources(self):
		return {"MenuText": "My Command",
			"Accel": "Ctrl+M",
			"ToolTip": "My extraordinary command",
			"Pixmap"  : """
			/* XPM */
			static const char *test_icon[]={
			"16 16 2 1",
			"a c #000000",
			". c None",
			"................",
			"................",
			"..############..",
			"..############..",
			"..############..",
			"......####......",
			"......####......",
			"......####......",
			"......####......",
			"......####......",
			"......####......",
			"......####......",
			"......####......",
			"......####......",
			"................",
			"................"};
			"""}

	def IsActive(self):
		if FreeCAD.ActiveDocument == None:
			return False
		else:
			return True

	def Activated(self):
		# do something here...

FreeCADGui.addCommand('MyCommand1',MyTool())
```

- La méthode **GetResources()** doit retourner un dictionnaire avec les attributs visuels de votre outil. **Accel**, définit une touche de raccourci, mais, n'est pas obligatoire.
- **IsActive()** définit si la commande est active, ou grisée dans les menus, et, barres d'outils.
- La méthode **Activated()** est exécutée lorsque la commande est appelée par un bouton de la barre d'outils, ou dans le menu, ou même par le script.

Création d'une commande FreeCAD en C++

Bientôt documentée.

< précédent: FreeCAD Build Tool Index suivant: Debugging >

Premiers tests

Avant de passer à la douloureuse phase de débogage, utilisez le framework de tests, pour vérifier, si les tests standards fonctionnent correctement. Si ce n'est pas le cas, c'est peut-être dû a une installation défectueuse.

Ligne de commande

Le débogage de FreeCAD est supporté par quelques mécanismes internes. La version en ligne de commande de FreeCAD fournit des options d'aide au débogage :

-v
: Avec l'option **"v"**, FreeCAD donne une sortie plus verbeuse (plus documentée).

-l
: Avec l'option **"l"**, FreeCAD écrit des informations supplémentaires dans un fichier **.log**.

These are the currently recognized options in FreeCAD 0.15:

Generic options:

```
-v [ --version ]      Prints version string
-h [ --help ]         Prints help message
-c [ --console ]      Starts in console mode
--response-file arg   Can be specified with '@name', too
```

Configuration:

```
-l [ --write-log ]       Writes a log file to:
                         /home/graphos/.FreeCAD/FreeCAD.log
--log-file arg           Unlike to --write-log this allows to log to an
                         arbitrary file
-u [ --user-cfg ] arg    User config file to load/save user settings
-s [ --system-cfg ] arg  Systen config file to load/save system settings
-t [ --run-test ] arg    Test level
-M [ --module-path ] arg Additional module paths
-P [ --python-path ] arg Additional python paths
```

Generating a Backtrace

If you are running a version of FreeCAD from the bleeding edge of the development curve, it may "crash". You can help solve such problems by providing the developers with a "backtrace". To do this, you need to be running a "debug build" of the software. "Debug build" is a parameter that is set at compile time, so you'll either need to compile FreeCAD yourself, or obtain a pre-compiled "debug" version.

For Linux

Prerequisites:

- software package gdb installed
- a debug build of FreeCAD
- a FreeCAD model that causes a crash

Steps: Enter the following in your terminal window:

```
$ cd FreeCAD/bin
$ gdb FreeCAD
```

GNUdebugger will output some initializing information. The (gdb) shows GNUDebugger is running in the terminal, now input:

```
(gdb) handle SIG33 noprint nostop
(gdb) run
```

FreeCAD will now start up. Perform the steps that cause FreeCAD to crash or freeze, then enter in the terminal window:

```
(gdb) bt
```

This will generate a lengthy listing of exactly what the program was doing when it crashed or froze. Include this with your problem report.

Python Debugging

Here is an example of using winpdb inside FreeCAD:

1. Run winpdb and set the password (e.g. test)
2. Create a Python file with this content

```
import rpdb2
rpdb2.start_embedded_debugger("test")
import FreeCAD
import Part
import Draft
print "hello"
print "hello"
import Draft
points=[FreeCAD.Vector(-3.0,-1.0,0.0),FreeCAD.Vector(-2.0,0.0,0.0)]
Draft.makeWire(points,closed=False,face=False,support=None)
```

1. Start FreeCAD and load the above file into FreeCAD
2. Press F6 to execute it
3. Now FreeCAD will become unresponsive because the Python debugger is waiting
4. Switch to the Windpdb GUI and click on "Attach". After a few seconds an item "<Input>" appears where you have to double-click
5. Now the currently executed script appears in Winpdb.
6. Set a break at the last line and press F5
7. Now press F7 to step into the Python code of Draft.makeWire

< précédent: Module Creation Index suivant: Testing >

FreeCAD est livré avec un vaste cadre de test. Les tests de bases sont basés, sur un ensemble de scripts **Python**, qui sont situées dans le module **test** (....FreeCAD.../Mod/Test).

Introduction

This is the list of test apps as of 0.15 Git 4207:

TestAPP.All

Add test function

BaseTests

Add test function

UnitTests

Add test function

Document

Add test function

UnicodeTests

Add test function

MeshTestsApp

Add test function

TestSketcherApp

Add test function

TestPartApp

Add test function

TestPartDesignApp

Add test function

Workbench

Add test function

Menu

Add test function

Menu.MenuDeleteCases

Add test function

Menu.MenuCreateCases

Add test function

< précédent: Debugging Index suivant: Branding >

Cet article décrit l'**image de marque de FreeCAD**. **Branding**, est le moyen de lancer votre propre application, sur les bases de **FreeCAD**.

Cela ne concerne que votre propre exécutable, ou, votre écran de démarrage (splash screen) ou jusqu'à ce que le programme complet soit retravaillé (refonte totale).

Grâce aux bases très souples de l'architecture de FreeCAD, il est très facile de l'utiliser, comme fondation pour votre programme personnalisé, ou pour une utilisation spécifique.

Generalités

La plupart des marques (branding) se font dans **MainCmd.cpp**, ou, **MainGui.cpp**. Ces projets génèrent les fichiers exécutables de **FreeCAD**.

Pour faire votre propre marque (branding), il suffit de copier **Main** (les projets principaux) ou **MainGui** (les projets graphiques GUI), et donner à l'exécutable un nom qui vous est propre, pour notre exemple, **FooApp.exe**. Les paramètres les plus importants pour un nouveau look, ne peuvent être fait qu'en un seul endroit, dans la **fonction main()**.

Voici la section de code qui contrôle la marque (branding) :

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

  // set the banner (for loging and console)
  App::Application::Config()["CopyrightInfo"] = sBanner;
  App::Application::Config()["AppIcon"] = "FooAppIcon";
  App::Application::Config()["SplashScreen"] = "FooAppSplasher";
  App::Application::Config()["StartWorkbench"] = "Part design";
  App::Application::Config()["HiddenDockWindow"] = "Property editor";
  App::Application::Config()["SplashAlignment" ] = "Bottom|Left";
  App::Application::Config()["SplashTextColor" ] = "#000000"; // black

  // Inits the Application
  App::Application::Config()["RunMode"] = "Gui";
  App::Application::init(argc,argv);

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

  Gui::Application::initApplication();
  Gui::Application::runApplication();
  App::Application::destruct();
```

```
  return 0;
}
```

La première entrée, **::Config** définit le nom du programme ici, **"FooApp.exe"**. Ce n'est pas le nom de l'exécutable qui peut être modifié en le renommant, ou par les paramètres du compilateur, mais le nom qui est affiché dans la barre des tâches sur les fenêtres, ou dans la liste des programmes sur les systèmes Unix.

Les lignes suivantes définissent les entrées de configuration de votre application **"FooApp"**, une description de la configuration, et de ses entrées, que vous trouverez dans **Start up and Configuration**.

Images

Image resources are compiled into FreeCAD using Qt's resource system (http://qt-project.org/doc/qt-4.8/resources.html). Therefore you have to write a .qrc file, an XML-based file format that lists image files on the disk but also any other kind of resource files. To load the compiled resources inside the application you have to add a line

```
Q_INIT_RESOURCE(FooApp);
```

into the main() function. Alternatively, if you have an image in XPM format you can directly include it into your main.cpp and add the following line to register it:

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

Branding XML

In FreeCAD there is also a method supported without writing a customized main() function. For this method you must write a file name called branding.xml and put it into the installation directory of FreeCAD. Here is an example with all supported tags:

```
<?xml version="1.0" encoding="utf-8"?>
<Branding>
       <Application>FooApp</Application>
       <WindowTitle>Foo App in title bar</WindowTitle>
       <BuildVersionMajor>1</BuildVersionMajor>
       <BuildVersionMinor>0</BuildVersionMinor>
       <BuildRevision>1234</BuildRevision>
       <BuildRevisionDate>2014/1/1</BuildRevisionDate>
       <CopyrightInfo>(c) My copyright</CopyrightInfo>
       <MaintainerUrl>Foo App URL</MaintainerUrl>
       <ProgramLogo>Path to logo (appears in bottom right corner)</ProgramLogo>
       <WindowIcon>Path to icon file</WindowIcon>
       <ProgramIcons>Path to program icons</ProgramIcons>
       <SplashScreen>splashscreen.png</SplashScreen>
       <SplashAlignment>Bottom|Left</SplashAlignment>
       <SplashTextColor>#ffffff</SplashTextColor>
       <SplashInfoColor>#c8c8c8</SplashInfoColor>
       <StartWorkbench>PartDesignWorkbench</StartWorkbench>
</Branding>
```

All of the listed tags are optional.

< précédent: Testing Index suivant: Localisation >

Localisation en général, est le processus de fourniture d'un logiciel avec une interface utilisateur (**GUI**) en plusieurs langues. Dans FreeCAD vous pouvez définir la langue d'interface utilisateur sous l'application **Edition → Préférences → Général → Onglet Général → général → Langue → Changer la langue**. FreeCAD utilise Qt (http://fr.wikipedia.org/wiki/Qt) pour activer le support de plusieurs langues. Sur les systèmes Unix/Linux, FreeCAD utilise les paramètres régionaux actuels de votre système par défaut.

Aider à la traduction de FreeCAD

Une des choses les plus très importantes que vous pouvez faire pour FreeCAD, si vous n'êtes pas programmeur, est de porter votre aide, pour traduire le programme dans votre langue. Pour ce faire, c'est maintenant très facile, avec la collaboration de Crowdin (http://crowdin.net) et l'utilisation de son système de traduction en ligne.

Comment traduire ?

- Aller à la page du projet de traduction de FreeCAD (http://crowdin.net/project/freecad) sur crowdin ;
- Connectez-vous en créant un nouveau profil, ou, en utilisant un compte tiers, comme votre adresse GMail;
- Cliquez sur la langue à laquelle vous souhaitez travailler ;
- Commencez la traduction en cliquant sur le bouton Traduire à côté des fichiers. Par exemple, **FreeCAD.ts** contient les chaînes de texte pour l'interface principale de FreeCAD .
- Vous pouvez opter pour les traductions existantes, ou vous pouvez créer une nouvelle langue.

PS : Si vous prenez une part active dans la traduction de FreeCAD, et, que vous voulez être informé ava

il est donc temps de revoir votre traduction, dans ce cas, s'il vous plaît abonnez vous sur : la page dé

Traduire avec Qt Linguist (ancienne méthode)

The following information doesn't need to be used anymore and will likely become obsolete. [afficher]

It is being kept here so that programmers may familiarize themselves with how it works.

Préparer vos propres modules ou applications pour la traduction

Prérequis

Pour localiser les modules d'applications dont vous avez besoin pour Qt, vous pouvez les télécharger à partir du site Web de Trolltech (http://www.trolltech.com/products/qt/downloads), mais ils sont également contenues dans le LibPack :

qmake
: Génère les fichiers du projet

lupdate
: ou mises à jour des textes originaux dans votre projet, par l'analyse du code source.

Qt-Linguist
: Le **Qt-Linguist** est très facile à utiliser et vous permet de faire votre traduction avec d'intéressantes fonctionnalités comme, un livre d'expressions pour les phrases communes.

Configuration d'un projet

Pour commencer la localisation de votre projet, visitez le **GUI-Part** du module et tapez à la ligne de commande :

```
qmake -project
```

Ici, le scan de votre répertoire "projet", contenant les fichiers textes, un fichier de projet est créé, comme dans l'exemple suivant :

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

Vous devez ajouter ces fichiers manuellement. La section `TRANSLATIONS` contient une liste de fichiers traduits pour chaque langue. Dans les exemples ci dessous, *MyMod_de.ts* est la traduction allemande (de).

Maintenant, exécutez `lupdate` pour extraire les chaines dans votre (GUI). Exécuter `lupdate` pendant un changement de code, est sans danger, car il ne supprime **jamais** de chaine de votre traduction. Mais ajoute seulement les nouvelles chaines traduites.

Maintenant, vous devez ajouter les fichiers `.ts` à votre projet VisualStudio. Précisez l'usage suivant pour leurs méthodes de constructions :

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

PS: Entrez ceci en ligne de commande, (le saut de ligne n'est là que pour la clarté).

En compilant le fichier `.ts` de l'exemple ci dessous, l'entête du fichier **MyMod_de.h** est créé. Le meilleur endroit pour l'inclure n'est pas dans le **App<Modul>Gui.cpp**. Dans notre exemple, le mieux serait, **AppMyModGui.cpp** .
Puis ajoutez la ligne:

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

pour publier votre traduction dans l'application.

Mise en place des fichiers Python pour la traduction

Pour faciliter la localisation des fichiers **.py** vous pouvez utiliser l'outil **"pylupdate4"** qui accepte un ou plusieurs fichiers **.py**. Avec l'option **-ts**, vous pouvez préparer ou mettre à jour un ou plusieurs fichiers **.ts**. Par exemple, pour préparer un fichier **.ts** pour le français, il suffit d'entrer à la ligne de commande :

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

L'outil **pylupdate** va scanner vos fichiers fonctions **.py** pour **translate()** ou **tr()** et créer un fichier **YourModule_fr.ts**.
Ce fichier peut être traduit avec **QLinguist** et un fichier **YourModule_fr.qm** produit à partir de **QLinguist**
ou avec la commande :

```
lrelease YourModule_fr.ts
```

Méfiez-vous de l'outil **pylupdate4** car il n'est pas très bon pour reconnaître la fonction **translate()**, il a besoin d'avoir une forme très spécifique (voir les fichiers Draft module comme exemple). A l'intérieur de votre dossier, vous pouvez alors configurer un traducteur comme celui-ci, (après avoir chargé votre QApplication mais, **AVANT** la création de n'importe quel widget qt) :

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

Optionnellement, vous pouvez également créer le fichier **XML** **Draft.qrc** avec ce contenu :

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

et démarrez **pyrcc4 Draft.qrc -o**, qrc_Draft.py crée une gros fichier Python, contenant toutes les ressources. D'ailleurs, cette approche fonctionne aussi pour mettre les fichiers icônes dans un fichier ressources.

Traduire le wiki

Ce wiki est l'hôte d'un très grand contenu. Le mis à jour, et, d'intéressantes informations sont rassemblées dans le manuel .

Ainsi, la première étape consiste à vérifier si la traduction manuelle a déjà été démarréé pour votre langue (regardez dans la barre latérale gauche, sous "manual").

Plugin de traduction

When the Wiki moved away from SourceForge, Yorik installed a Translation plugin (http://www.mediawiki.org /wiki/Help:Extension:Translate) which allows to ease translations between pages. For example, the page title can now be translated. Other advantages of the Translation plugin are that it keeps track of translations, notifies if the original page has been updated, and maintains translations in sync with the original English page.

The tool is documented in Extension:Translate (http://www.mediawiki.org/wiki/Help:Extension:Translate), and is part of a Language Extension Bundle (http://www.mediawiki.org /wiki/MediaWiki_Language_Extension_Bundle).

To quickly get started on preparing a page for translation and activating the plugin, please read the Page translation example (http://www.mediawiki.org/wiki/Help:Extension:Translate /Page_translation_example).

To see an example of how the Translation tool works once the translation plugin is activated on a page, you can visit the Main Page. You will see a new language menu bar at the bottom. It is automatically generated. Click for instance on the German link, it

will get you to Main Page/de. Right under the title, you can read "This page is a **translated version** of a page Main Page and the translation is xx% complete." (xx being the actual percentage of translation). Click on the "translated version" link to start translation, or to update or correct the existing translation.

You will notice that you cannot directly edit a page anymore once it's been marked as a translation. You have to go through the translation utility.

When adding new content, the English page should be created first, then translated into another language. If someone wants to change/add content in a page, he should do the English one first.

It is recommended to have basic knowledge of wiki style formatting and general guidelines of the FreeCAD wiki, because you will have to deal with some tags while translating. You can find this information on WikiPages.

The sidebar (navigation menu on the left) is also translatable. Please follow dedicated instructions on Localisation Sidebar page.

REMARK: The first time you switch a page to the new translation system, it looses all its old 'manual' translations. To recover the translation, you need to open an earlier version from the history, and copy/paste manually the paragraphs to the new translation system.

Remark: to be able to translate in the wiki, you must of course gain wiki edit permission.

If you are unsure how to proceed, don't hesitate to ask for help in the forum (http://forum.freecadweb.org).

Old translation instructions

These instructions are for historical background only, [afficher] while the pages are being passed to the new translation plugin.

< précédent: Branding Index suivant: Extra python modules >

Cette page contient plusieurs modules python supplémentaires ou d'autres bouts de code qui peuvent être téléchargés gratuitement sur Internet, et ajouter des fonctionnalités à votre installation de FreeCAD.

PySide (précédemment PyQt4)

- page officielle (PySide): http://qt-project.org/wiki/PySide
- licence: LGPL
- option, plusieurs modules sont nécessaires et d'autres modules peuvent être ajoutés : Draft, Arch, Ship, Plot, OpenSCAD, Spreadsheet

PySide (auparavant PyQt) est requise par tous les modules de FreeCAD et pour accéder à l'interface Qt de FreeCAD. Il est déjà livré dans dans les versions FreeCAD, et est généralement installé automatiquement par FreeCAD sur Linux, l'installation peut se faire à partir des dépôts officiels. Si ces modules (Draft, Arch, etc) sont activés après l'installation de FreeCAD, cela signifie que PySide (auparavant PyQt) est déjà installé, et vous n'avez pas besoin de faire quoi que ce soit de plus.

Remarque : PyQt4 va devenir progressivement obsolète dans FreeCAD, après la version 0.13, la préférence ira sur PySide (http://qt-project.org/wiki/PySide), qui fait exactement le même travail, mais dispose d'une licence (**LGPL**) plus compatible avec FreeCAD.

Installation

Linux

La façon la plus simple d'installer PySide est de l'installer par le biais du gestionnaire de paquets de votre distribution. Sur les systèmes Debian / Ubuntu, le nom du package est généralement **python-PySide**, tandis que sur les systèmes basés sur RPM il est nommé **Pyside**. Les dépendances nécessaires (Qt et SIP) seront pris en charge automatiquement.

Windows

Le programme peut être téléchargé à partir PySide Downloads (http://qt-project.org /wiki/Category:LanguageBindings::PySide::Downloads). Vous aurez besoin d'installer les bibliothèques Qt et SIP avant d'installer PySide (à documenter).

MacOSX

PyQt pour Mac doit être installé via homebrew ou port. Pour plus d'informations voir CompileOnMac/fr#Dépendances_de_l'installation Dépendances_de_l'installation.

Utilisation

Une fois installé, vous pouvez vérifier le bon fonctionne de l'installation, en tapant dans la console **Python** de FreeCAD :

```
import PySide
```

Pour accéder à l'interface de FreeCAD, tapez :

```
from PySide import QtCore,QtGui
FreeCADWindow = FreeCADGui.getMainWindow()
```

Maintenant, vous pouvez commencer l'exploration de l'interface avec la commande **dir()**. Vous pouvez ajouter de nouveaux éléments, comme un widget personnalisé, avec des commandes comme :

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

Travailler avec Unicode :

```
text = text.encode('utf-8')
```

Travailler avec QFileDialog et OpenFileName :

```
path = FreeCAD.ConfigGet("AppHomePath")
#path = FreeCAD.ConfigGet("UserAppData")
OpenName, Filter = PySide.QtGui.QFileDialog.getOpenFileName(None, "Read a txt file", path, "*.txt")
```

Travailler avec QFileDialog et SaveFileName :

```
path = FreeCAD.ConfigGet("AppHomePath")
#path = FreeCAD.ConfigGet("UserAppData")
SaveName, Filter = PySide.QtGui.QFileDialog.getSaveFileName(None, "Save a file txt", path, "*.txt")
```

Exemple de transition de PyQt4 vers PySide

PS: ces exemples d'erreurs ont été trouvées dans la transition de PyQt4 à PySide et ces corrections ont été faites, d'autres solutions sont certainement disponibles avec les exemples ci-dessus

```
try:
    import PyQt4                                    # PyQt4
    from PyQt4 import QtGui ,QtCore                 # PyQt4
    from PyQt4.QtGui import QComboBox               # PyQt4
    from PyQt4.QtGui import QMessageBox             # PyQt4
    from PyQt4.QtGui import QTableWidget, QApplication  # PyQt4
    from PyQt4.QtGui import *                       # PyQt4
    from PyQt4.QtCore import *                      # PyQt4
except Exception:
    import PySide                                   # PySide
    from PySide import QtGui ,QtCore                # PySide
    from PySide.QtGui import QComboBox              # PySide
    from PySide.QtGui import QMessageBox            # PySide
    from PySide.QtGui import QTableWidget, QApplication # PySide
    from PySide.QtGui import *                      # PySide
    from PySide.QtCore import *                     # PySide
```

Pour accéder à l'interface FreeCAD, tapez: Vous pouvez ajouter de nouveaux éléments, comme un widget personnalisé, avec des commandes comme :

```
myNewFreeCADWidget = QtGui.QDockWidget()          # create a new dockwidget
myNewFreeCADWidget.ui = Ui_MainWindow()           # myWidget_Ui()             # load the Ui script
myNewFreeCADWidget.ui.setupUi(myNewFreeCADWidget) # setup the ui
try:
    app = QtGui.qApp                              # PyQt4 # the active qt window, = the freecad window s
    FCmw = app.activeWindow()                     # PyQt4 # the active qt window, = the freecad window s
    FCmw.addDockWidget(QtCore.Qt.RightDockWidgetArea,myNewFreeCADWidget) # add the widget to the main wi
except Exception:
    FCmw = FreeCADGui.getMainWindow()             # PySide # the active qt window, = the freecad window s
    FCmw.addDockWidget(QtCore.Qt.RightDockWidgetArea,myNewFreeCADWidget) # add the widget to the main wi
```

Travailler avec Unicode :

```
try:
    text = unicode(text, 'ISO-8859-1').encode('UTF-8')  # PyQt4
except Exception:
    text = text.encode('utf-8')                         # PySide
```

Travailler avec QFileDialog et OpenFileName :

```
OpenName = ""
try:
    OpenName = QFileDialog.getOpenFileName(None,QString.fromLocal8Bit("Lire un fichier FCInfo ou txt"),pa
except Exception:
    OpenName, Filter = PySide.QtGui.QFileDialog.getOpenFileName(None, "Lire un fichier FCInfo ou txt", pa
```

Travailler avec QFileDialog et SaveFileName :

```
SaveName = ""
try:
    SaveName = QFileDialog.getSaveFileName(None,QString.fromLocal8Bit("Sauver un fichier FCInfo"),path,"
except Exception:
    SaveName, Filter = PySide.QtGui.QFileDialog.getSaveFileName(None, "Sauver un fichier FCInfo", path,
```

Travailler avec MessageBox:

```
def errorDialog(msg):
    diag = QtGui.QMessageBox(QtGui.QMessageBox.Critical,u"Error Message",msg )
    try:
        diag.setWindowFlags(PyQt4.QtCore.Qt.WindowStaysOnTopHint) # PyQt4 # this function sets the windo
    except Exception:
        diag.setWindowFlags(PySide.QtCore.Qt.WindowStaysOnTopHint)# PySide # this function sets the windo
#    diag.setWindowModality(QtCore.Qt.ApplicationModal)       # function has been disabled to promote "W
    diag.exec_()
```

Travailler avec setProperty (PyQt4) et setValue (PySide)

```
self.doubleSpinBox.setProperty("value", 10.0)  # PyQt4
```

remplacer par :

```
self.doubleSpinBox.setValue(10.0)  # PySide
```

Travailler avec setToolTip

```
self.doubleSpinBox.setToolTip(_translate("MainWindow", "Coordinate placement Axis Y", None))  # PyQt4
```

remplacer par :

```
self.doubleSpinBox.setToolTip(_fromUtf8("Coordinate placement Axis Y"))  # PySide
```

ou

```
self.doubleSpinBox.setToolTip(u"Coordinate placement Axis Y.")# PySide
```

Documentation

Plus de tutoriels sur **PyQt4** (y compris sur la façon de construire des interfaces avec **Qt Designer** pour utiliser avec python) :

- API PyQt4 (http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/classes.html) - La référence officielle sur l'**API de PyQt4**
- Introduction PyQt4 (http://www.rkblog.rk.edu.pl/w/p/introduction-pyqt4/)- une simple introduction.
- un tutoriel (http://www.zetcode.com/tutorials/pyqt4/) - vraiment complet.

Pivy

- homepage: https://bitbucket.org/Coin3D/coin/wiki/Home
- license: BSD
- option, utilisé par tous les modules de FreeCAD: Draft, Arch

Pivy a besoin de plusieurs modules pour accéder à la vue 3D de FreeCAD. Pour les fenêtres, pivy est déjà fourni dans l'installateur de FreeCAD pour Linux, il est généralement installé automatiquement lorsque vous installez FreeCAD partir d'un référentiel officiel. Sur MacOSX, malheureusement, vous aurez besoin de compiler Pivy vous même.

Installation

Prérequis

Je crois, qu'avant de compiler **Pivy (http://pivy.coin3d.org/)** vous devez avoir **Coin (http://www.coin3d.org/)** et **SoQt (http://www.coin3d.org/lib/soqt/releases/1.5.0)** d'installés.

J'ai trouvé que pour la compilation sur **Mac**, il suffisait d'installer le **Coin3 binary package (http://www.coin3d.org /lib/plonesoftwarecenter_view)**.
La tentative d'installation de **Coin** sur **MacPorts** était problématique : j'ai essayé d'ajouter un grand nombre de paquets X Windows, et, finalement, tout c'est terminé avec une erreur de script !

Pour Fedora, j'ai trouvé un **RPM** avec **Coin3**.

SoQt, compilé à partir des sources (http://www.coin3d.org /lib/soqt/releases/1.5.0) fonctionne très bien sur **Mac et Linux**.

Debian & Ubuntu

Depuis **Debian Squeeze** et **Ubuntu Lucid**, **Pivy** est disponible directement à partir des dépôts officiels, et, nous permet d'économiser beaucoup de tracas.
En attendant, vous pouvez soit télécharger l'un des **packages** que nous avons fait (pour Debian et Ubuntu karmic), disponibles sur les pages de téléchargements , ou, vous pouvez le compiler vous-même.

La meilleure façon de compiler facilement **Pivy**, est de prendre le **debian source package** pour **Pivy**, et, faire un **package** avec **debuild**.
C'est le même code source que sur le site officiel de **Pivy**, mais, les gens de **Debian** ont ajoutés plusieurs bug-fixing. Il compile également très bien sur : **Ubuntu Karmic (http://packages.debian.org/squeeze/python-pivy)** ...
télécharger **.orig.gz** et **.diff.gz**, décompressez le tout, puis appliquez **.diff** à la source :
allez dans le dossier source de **Pivy** décompressé, et appliquez le patch **.diff** :

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

alors

```
debuild
```

pour avoir **Pivy**, correctement compilé, avec un package officiellement installable. Ensuite, il suffit d'installer le package avec **gdebi**.

Autres distributions Linux

D'abord, téléchargez les dernières sources du project's repository (http://pivy.coin3d.org/mercurial/) :

```
hg clone http://hg.sim.no/Pivy/default Pivy
```

En Mars 2012, la dernière version était la **pivy-0.5**.

Ensuite, vous avez besoin d'un outil appelé **SWIG** pour générer le code C++ pour les **Python bindings**. **Pivy-0.5** rapports qui a été testé seulement avec **SWIG 1.3.31, 1.3.33, 1.3.35 et 1.3.40**. Ainsi, vous pouvez télécharger une archive source pour l'une de ces anciennes versions de SWIG (http://www.swig.org).
Puis, décompressez-le, et, faites en ligne de commande (en tant que root) :

```
./configure
make
make install (or checkinstall if you use it)
```

Il faut quelques secondes pour la compilation.

Alternativement, vous pouvez essayer avec une compilation plus récent **SWIG**. En Mars 2012, la version référentielle typique était **2.0.4**.
Pivy a un problème de compilation avec les versions inférieures **2.0.4** de **SWIG** sur Mac OS (voir ci-dessous), mais semble compiler correctement sur **Fedora Core 15**.

Après cela, allez dans le source **Pivy** et tapez :

```
python setup.py build
```

pour créer les fichiers sources. Notez que cette génération de fichiers peut produire des milliers de mises en garde, mais j'espère qu'il n'y aura pas d'erreurs.

Ceci est probablement obsolète, mais vous risquez de rencontrer une erreur de compilation, ou, un **"const char*"** ne peut pas être converti en un **"char*"**.
Pour corriger cela, il vous suffit d'écrire une **"const"**, dans les lignes appropriées, avant la génération. Il y a six lignes à corriger.

Après cela, installez (en tant que root) :

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

Ça y est, pivy est installé.

Mac OS

Ces instructions peuvent ne pas être complètes. Quelque chose plus ou moins comme cela a fonctionné pour **OS 10.7 de Mars 2012**. J'utilise **MacPorts (http://www.macports.org/)** pour les dépôts, mais d'autres options devraient également fonctionner.

En ce qui concerne linux, téléchargez les dernières sources :

```
hg clone http://hg.sim.no/Pivy/default Pivy
```

Si vous n'avez pas **hg**, vous pouvez l'obtenir à partir **MacPorts (http://www.macports.org/)** :

```
port install mercurial
```

Puis, comme ci-dessus vous avez besoin **SWIG (http://www.swig.org/)**.

Faites :

```
port install swig
```

J'ai trouvé que j'avais besoin aussi de faire :

```
port install swig-python
```

En Mars 2012, **MacPorts SWIG** est la version **2.0.4**. Comme il est indiqué ci-dessus pour Linux, il vaudrait mieux télécharger une version plus ancienne. **SWIG 2.0.4** semble avoir un bug qui empêche la compilation de **Pivy**.
Regardez le premier message dans ce : *digest (https://sourceforge.net/mailarchive /message.php?msg_id=28114815)*

Cela peut être corrigé, en modifiant les 2 emplacements source et déréférencer : ***arg4, *arg5** à la place de **arg4, arg5**.
Maintenant nous pouvons compiler **Pivy**:

```
python setup.py build
sudo python setup.py install
```

Windows

En supposant que vous utilisiez **Visual Studio 2005** ou une version ultérieure, vous devrez ouvrir une invite de commande avec **Visual Studio 2005 Command prompt** dans le menu Outils.
Si l'interpréteur **Python** n'est pas encore dans le chemin système (PATH), faites :

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

Pour que **Pivy** soit fonctionnel, vous devriez télécharger les dernières sources à partir du référentiel du projet :

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

Ensuite, vous avez besoin d'un outil appelé SWIG (http://www.swig.org/) pour générer le code C++ pour les **Python bindings**. Il est recommandé d'utiliser la version **1.3.25 de SWIG**, pas la dernière version, parceque, **Pivy** ne fonctionne pas correctement avec la version **1.3.25**. Télécharger le binaire pour la version **1.3.25** de **Swig (http://www.swig.org)**.
Puis décompressez-le et à partir de la ligne de commande, ajoutez le chemin (path) du système

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

et définir le chemin approprié à **COINDIR** :

```
set COINDIR=path_to_coin
```

Sous Windows, le fichier de configuration **Pivy** attend **SoWin** au lieu de **SoQt** par défaut. Je n'ai pas trouvé de façon évidente pour compiler avec **SoQt**, alors, j'ai modifié le fichier **setup.py** directement.
A la ligne 200 il suffit de retirer la partie **sowin** : **('gui._sowin', 'sowin-config', 'pivy.gui.') (ne pas enlever la parenthèse fermante !)**.

Après cela, allez dans le source de **pivy** et tapez :

```
python setup.py build
```

qui crée les fichiers source. Vous pouvez rencontrer une erreur de compilation, cause, **plusieurs fichiers d'en-tête n'ont pas été trouvés**.
Dans ce cas, réglez la variable **INCLUDE** comme ceci :

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

et si les en-têtes **soqt**, ne sont pas au même endroit que les en-têtes **Coin**, faites aussi ceci :

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

et finalement, pour les en-têtes **Qt** faites :

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

Si vous utilisez **Express Edition of Visual Studio**, vous pouvez obtenir une exception **Python keyerror**.
Dans ce cas, vous devez modifier de petites choses dans **msvccompiler.py**, qui se trouve, dans votre installation Python.

Aller à la ligne 122 et remplacez la ligne :

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

par

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

Puis réessayez.
Si vous obtenez une deuxième erreur comme :

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

vous devez également remplacer la ligne 128 comme ceci :

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

par

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

Réessayez encore une fois.
Si vous obtenez de nouveau une erreur comme :

```
error: Python was built with Visual Studio version 8.0, and extensions need to be built with the same ve
```

alors vous devriez vérifier les variables d'environnement **DISTUTILS_USE_SDK** et **MSSDK** avec :

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

Si ce n'est pas toujours pas arrangé, il suffit de définir à 1 :

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

Maintenant, vous pouvez rencontrer une erreur de compilation, ou un **const char*** ne peut pas être converti en un **char***.
Pour corriger cela il vous suffit d'écrire un **const** avant, dans les lignes appropriées, il y a six lignes à corriger.
Après copiez le répertoire généré par **Pivy** dans un endroit où l'interpréteur **Python** de FreeCAD peut le trouver.

Utilisation

Pour vérifier si pivy est correctement installé :

```
import pivy
```

Pour avoir accès à Pivy à partir de la scénographique de FreeCAD, procédez comme ceci:

```
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 "c
FCSceneGraph.addChild(coin.SoCube()) # add a box to scene
```

Vous pouvez maintenant explorer la **FCSceneGraph** avec la commande **dir()**.

Documentation

Malheureusement, la documentation sur **Pivy** est "pour le moment" presque inexistante sur le net. Mais vous pouvez trouver de la documentation très utile sur **Coin**, car **Pivy** a simplement traduit les fonctions, **Coin**, des nœuds et des méthodes en **Python**, les noms sont conservés (mêmes noms) ainsi que les propriétés ne sont différentes que par la syntaxe

entre le **C** et **Python** :

- https://bitbucket.org/Coin3D/coin/wiki/Documentation - Coin3D API Reference
- http://www-evasion.imag.fr/~Francois.Faure /doc/inventorMentor/sgi_html/index.html - The Inventor Mentor - La "bible" de Inventor langage de description de scène.

Vous pouvez également consulter le fichier **Draft.py** dans le dossier **FreeCAD Mod/Draft**, car **Pivy** est fortement utilisé.

pyCollada

- homepage: http://pycollada.github.com
- license: BSD
- option, est nécessaire pour permettre l'importation et l'exportation de fichiers Collada (.DAE)

pyCollada (http://pycollada.github.com) est une bibliothèque **Python** qui permet aux programmes de lire et d'écrire des fichiers **Collada (*.DAE) (http://en.wikipedia.org /wiki/COLLADA)**. Lorsque **pyCollada** est installé sur votre système, FreeCAD (available in version 0.13

) le détecte et ajoute les options d'importation et d'exportation, qui permettent l'ouverture et l'enregistrement de fichiers au format **Collada**.

Installation

Pycollada n'est généralement pas encore disponible dans les dépôts des distributions Linux, mais puisqu'il est fait uniquement en **Python**, il ne nécessite pas de compilation, et est facile à installer.
Vous avez 2 façons de l'installer, soit directement à partir du **pycollada git repository** officiel, ou avec l'outil **easy_install**.

Linux

Dans les deux cas, vous aurez besoin des paquetages suivants, installés d'avance sur votre système :

```
python-lxml
python-numpy
python-dateutil
```

Depuis le dépôt git (pycollada git repository)

```
git clone git://github.com/pycollada/pycollada.git pycollada
cd pycollada
sudo python setup.py install
```

Avec easy_install (easy_install)

En supposant que vous avez déjà installé complètement **Python**, l'utilitaire **easy_install** doit être déjà présent :

```
easy_install pycollada
```

Vous devez vous assurer que **pycollada**, est correctement installé, en utilisant la commande suivante dans la console Python :

```
import collada
```

Si la commande ne retourne aucun message d'erreur, alors tout est OK.

Windows

1. Installez Python. Alors que FreeCAD et quelques autres programmes sont livrés avec une version embarquée de Python, une installation fixe aidera les prochaines étapes. Vous pouvez obtenir Python ici: https://www.python.org /downloads/. Bien sûr, vous devrez choisir la bonne version, dans ce cas, ce serait 2.6.X, FreeCAD utilise actuellement la 2.6.2 (Personnellement je suis installé avec la version 2.6.2, et pour la forme, vous pouvez vérifier la version en démarrant

Python.exe dans le dossier bin de FreeCAD). Vous aurez également à ajouter le chemin du répertoire d'installation dans la variable path afin que vous puissiez accéder à Python à partir de la console (cmd). Maintenant, nous pouvons installer tout ce qu'il nous manquante, au total il y a trois choses que nous devons installer: numpy, setuptools et pycollada
2. Fetch numpy ici: http://sourceforge.net/projects/numpy/files/NumPy/. Choisissez une version qui s'adapte à la version utilisée par FreeCAD, dans chaque dossier de version numpy il existe plusieurs programmes d'installation pour les différentes versions de Python, l'installateur sera placé dans le dossier numpy de votre installation Python, où FreeCAD peut y accéder aussi
3. Fetch setuptools ici : https://pypi.python.org/pypi/setuptools (Nous devons installer les setuptools pour installer pycollada dans l'étape suivante)
4. décompressez dans un dossier le fichier setuptools téléchargé
5. Démarrer une console (cmd) avec la permission admin
6. Accédez au dossier décompressé de setuptools
7. installer les setuptools "Python setup.py install" par basculement dans la console (cmd), ne fonctionnera pas si Python n'est pas installé ou lorsque la variable path n'a pas été configurée
8. Fetch pycollada ici: https://pypi.python.org/pypi/pycollada/ (a déjà été affiché ci-dessus) et encore une fois:
9. Décompressez le fichier pycollada téléchargé dans un dossier
10. Démarrer une console (cmd) avec la permission d'administration, ou utilisez celui que vous avez ouvert il n'y a pas longtemps
11. Accédez au dossier pycollada décompressé
12. Installez les setuptools "Python setup.py install" à partir de la console (cmd)

- Une autre référence pour utiliser easy_install: http://jishus.org/?p=452

Mac OS

Si vous utilisez l'accumulation des Homebrew FreeCAD vous

pouvez installer pycollada dans votre système Python en utilisant pip.

Si vous devez installer pip:

```
$ sudo easy_install pip
```

Installer pycollada:

```
$ sudo pip install pycollada
```

Si vous utilisez une version binaire de FreeCAD, vous pouvez dire pip installez pycollada dans le site-packages à l'intérieur FreeCAD.app:

```
$ pip install --target="/Applications/FreeCAD.app/Contents/lib/python2.7/site-packages" pycollada
```

IfcOpenShell

- homepage: http://www.ifcopenshell.org
- license: LGPL
- option, requis pour étendre les capacités d'importation de fichiers IFC

IFCOpenShell, est une bibliothèque actuellement en développement, ce qui permet d'importer (et bientôt d'exporter) Industry foundation Classes (*.Fichiers IFC) (http://fr.wikipedia.org/wiki/Industry_Foundation_Classes).

Ceci est une extension pour le format STEP (http://fr.wikipedia.org/wiki/Standard_pour_l%27%C3%A9change_de_donn%C3%A9es_de_produit), et, devient la norme dans les workflows BIM (http://fr.wikipedia.org/wiki/Building_Information_Modeling). Lorsque **ifcopenshell** est correctement installé sur votre système, le Module Arch de FreeCAD le détectera, et, l'utilisera pour importer des fichiers **IFC**. Étant donné qu'**ifcopenshell** est basé sur OpenCasCade, comme FreeCAD, la qualité de l'importation est très élevée, en produisant une géométrie de solides de haute qualité.

Installation

Étant donné que '**ifcopenshell** est assez nouveau, vous devrez probablement le compiler vous-même.

Linux

Vous aurez besoin de deux ou trois paquets de développement, installés sur votre système afin de rassembler les ifcopenshell :

```
liboce-*-dev
python-dev
swig
```

mais, étant donné que FreeCAD exige tout, vous pouvez compiler FreeCAD, vous n'aurez aucune dépendance supplémentaire pour compiler IfcOpenShell.

Prenez le dernier code source ici :

```
svn co https://svn.code.sf.net/p/ifcopenshell/svn/trunk ifcopenshell ifcopenshell
```

or

```
svn co https://ifcopenshell.svn.sourceforge.net/svnroot/ifcopenshell ifcopenshell
```

Le processus de création est très simple :

```
mkdir ifcopenshell-build
cd ifcopenshell-build
cmake ../ifcopenshell/cmake
```

ou, si vous utilisez **oce** au lieu d'**opencascade** :

```
cmake -DOCC_INCLUDE_DIR=/usr/include/oce ../ifcopenshell/cmake
```

Étant donné que **ifcopenshell** est fait principalement pour Blender (http://www.blender.org/), il utilise **python3** par défaut. Pour l'utiliser à l'intérieur de FreeCAD, vous devez le compiler avec la même version de Python qui est utilisé dans FreeCAD.

Vous devrez peut-être forcer les paramètres avec la version de **Python** et **cmake** (réglez la version de Python avec la vôtre) :

```
cmake -DOCC_INCLUDE_DIR=/usr/include/oce -DPYTHON_INCLUDE_DIR=/usr/include/python2.7 -DPYTHON_LIBRARY=/us
```

Alors :

```
make
sudo make install
```

Vous pouvez vérifier que **ifcopenshell**, a été correctement installé en tapant dans la console Python :

```
import IfcImport
```

Si la commande ne retourne aucun message d'erreur, alors tout est OK.

Windows

Documentation copiée à partir du fichier README IfcOpenShell

Les utilisateurs sont priés d'utiliser le fichier **.sln** de Visual Studio qui se trouve dans **win/folder**.

Pour les utilisateurs de Windows une version pré-construite Open CASCADE est disponible sur le site d'OpenCascade (http://opencascade.org). Téléchargez, et, installez cette version dans le chemin d'accès d'Open CASCADE, et, des fichiers de la bibliothèque de MS Visual Studio C++.

Pour créer le **IfcPython wrapper**, **SWIG** doit être installé. Téléchargez la dernière version de swigwin (http://www.swig.org /download.html). Après avoir extrait le fichier **.zip**, veuillez ajouter le dossier à la variable **d'environnement PATH**. Python doit être installé, veuillez fournir les chemins d'accès des fichiers include, et, bibliothèque pour Visual Studio.

Teigha Converter

- homepage: http://www.opendesign.com/guestfiles/TeighaFileConverter
- license: freeware
- option, utilisé pour permettre l'importation et l'exportation de fichiers DWG

Le convertisseur Teigha Converter est un petit utilitaire disponible gratuitement qui permet de convertir plusieurs versions de fichiers DWG et DXF. FreeCAD peut l'utiliser pour permettre l'importation et l'exportation de fichiers DWG, en convertissant les fichiers DWG au format DXF de manière transparente, puis utiliser son importateur DXF standard pour importer le contenu du fichier. Les restrictions de la DXF importer s'appliquent.

Installation

S'installe sur toutes les plateformes, par l'installation du package approprié dans http://www.opendesign.com/guestfiles/TeighaFileConverter. Après l'installation, si l'utilitaire n'est pas trouvé automatiquement par FreeCAD, vous devrez configurer manuellement le chemin de l'exécutable du convertisseur, dans le menu Edition -> Préférences -> Projet -> Options d'importation/exportation.

< précédent: Localisation Index suivant: Source documentation >

Credits

<translate> FreeCAD would not be what it is without the generous contributions of many people. Here's an overview of the people and companies who contributed to FreeCAD over time. For credits for the third party libraries see the Third Party Libraries page.

Developement

Project managers

Lead developers of the FreeCAD project: </translate>

- Jürgen Riegel
- Werner Mayer
- Yorik van Havre

<translate>

Main developers

People who work regularly on the FreeCAD code: </translate>

- Logari81 (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=270)
- Luke A. Parry (http://freecadamusements.blogspot.co.uk/)
- Jose Luis Cercos Pita (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=574)
- Jan Rheinlaender (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=997)
- shoogen (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=765)
- tanderson69 (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=208)

<translate>

Other coders

People who contributed code to the FreeCAD project:

</translate>

- ickby (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=686)
- jmaustpc (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=611)
- j-dowsett (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=652)
- keithsloan52 (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=930)
- wandererfan (http://forum.freecadweb.org /memberlist.php?mode=viewprofile&u=1375)
- Joachim Zettler
- Graeme van der Vlugt
- Berthold Grupp
- Georg Wiora
- Martin Burbaum
- Jacques-Antoine Gaudin
- Ken Cline
- Dmitry Chigrin
- Remigiusz Fiedler (DXF-parser)

<translate>

Companies

Companies which donated code or developer time: </translate>

- Imetric 3D

<translate>

Community

People from the community who put a lot of efforts in helping the FreeCAD project either by being active on the forum, keeping a blog about FreeCAD, making video tutorials, packaging FreeCAD for Windows/Linux/MacOS X, writing a FreeCAD book... (listed by alphabetical order) </translate>

- bejant (http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1940)
- Brad Collette (http://www.packtpub.com/freecad-solid-modeling-with-python/book)
- cblt2l (http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=251)
- Daniel Falck (http://opensourcedesigntools.blogspot.com/)
- Eduardo Magdalena
- hobbes1069 (http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=725)
- jdurston (5needinput) (http://www.youtube.com/user/5needinput)
- jmaustpc (http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=611)
- John Morris (butchwax) (http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=861)
- Kwahooo (http://freecad-tutorial.blogspot.com/)
- lhagan (http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=108)
- marcxs (http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1047)
- Mario52
- Normandc
- peterl94 (http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1819)
- pperisin (http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=356)
- Quick61
- Renatorivo
- Rockn

<translate> </translate>

Catégories : Poweruser Documentation/fr | Python Code/fr | Tutorials/fr | Poweruser Documentation | Developer Documentation/fr | Developer

---

- Dernière modification de cette page le 8 février 2015 à 23:42.
- Cette page a été consultée 15 587 fois.
-