Bases de Tcl/Tk

Emmanuel Grolleau

grolleau@ensma.fr
http://www.lisi.ensma.fr/members/grolleau

Janvier 1998

Table des matières

I. Introduction

Tcl/Tk (prononcer Teackle Teakey à l'anglaise) est un langage interprété portable sur Mac, PC et station unix. Tcl (Tool Command Language) est un langage script non typé, et Tk (Tool Kit) est un ensemble d'outils permettant de construire des interfaces graphiques.

A. Pourquoi Tcl/Tk ?

- Tcl/Tk permet très simplement de construire des interfaces graphiques. Sa facilité d'utilisation réduit largement le temps de développement de l'interface graphique d'un logiciel (j'estime le temps de développement Interface/Noyau passé de 70%/30% à 30%/70% grâce à l'utilisation de Tcl/Tk).
- De plus, Tcl est suffisamment puissant pour développer une application de petite taille. Cependant pour des applications de moyenne et grande taille, Tcl/Tk permet un interfaçage simple avec le langage C. Pour les programmeurs Ada, adatcl permet d'utiliser Tcl/Tk avec Ada.
- La portabilité de Tcl/Tk permet de passer d'une plate-forme de développement à une autre (Mac, PC, station unix) sans aucune modification du programme! De plus, il est envisagé d'intégrer un interpréteur Tcl/Tk en natif dans certains systèmes d'exploitation.

Ce langage est donc en pleine expansion, portable, simple, gratuit, et il s'avère être un des langages les plus pratiques de développement d'interface graphique.

B. Philosophie générale

Tcl est un script interprété par la commande *tclsh*, il peut être vu comme une extension du shell unix *sh*. Ce script peut être lancé soit en mode interactif:
>tclsh
*% **set** a 1*
soit en mode autonome en lui donnant un fichier d'entrée:
>tclsh -f monfichier.tcl
Le premier cas sera utilisé pour le développement d'une application, alors que le deuxième cas permettra d'utiliser un programme tcl comme un programme à part entière.

Tk est une surcouche de Tcl, et est aussi un script interprété, mais par l'interpréteur *wish* (Widget shell) qui inclue *tclsh*. *Wish* permet la création d'objets graphiques (fenêtres, boutons, menus, canvas...), nommés *widgets*, ainsi que leur gestion, destruction, ...

Comme dans tout shell, une instruction doit s'écrire sur une seule ligne. On peut tout de même couper une ligne sur plusieurs pour plus de commodité, en faisant précéder les retours chariot d'un antislash (\). De plus un bloc, c'est à dire du code entre accolades, peut être écrit sur plusieurs lignes. Bien sûr, chaque instruction du bloc doit porter sur une seule ligne ou plusieurs avec \, ou être un bloc...

II. Programmation Tcl

A. Variables Tcl

En Tcl, les variables sont non typées, et peuvent être, suivant le cas, considérées comme chaînes de caractères (le cas le plus large), listes (ensemble ordonné d'éléments non typés qui peuvent aussi être eux mêmes des listes), entiers ou flottants. Elles peuvent aussi être des tableaux à une ou plusieurs dimensions. Une même variable peut être considérée de types différents suivant l'opération que l'on fait sur elle.
L'accès à une variable *nom_variable* se fait par *$nom_variable*. L'affectation se fait par la commande *set*.

```
% set a 12                    #la variable a reçoit "12"
% puts a                      #affichage du caractère "a"
a
% puts $a                     #affichage du contenu de la variable a
12
% set a b                     #la variable a reçoit la lettre "b"
% set b iletaitunebergere     #la variable b reçoit la chaîne "iletaitunebergere"
% puts $a                     #affichage de la valeur de a
b
% puts $b                     #affichage de la valeur de b
iletaitunebergere
% puts [subst $$a]            #affichage de la valeur de la variable contenue dans la variable a,
iletaitunebergere             #la fonction subst force l'évaluation le plus loin possible des
#variables. L'emploie des crochets sera expliqué plus loin
```

1. Chaînes de caractères pouvant contenir des espaces:

```
% set toto il etait une bergere          #utilisation de la fonction set avec trop d'arguments
wrong # args: should be "set varName ?newValue?"
% set toto "il etait une bergere"        #utilisation correcte de set
% set toto {il etait une bergere}        #autre utilisation correcte de set
% set a bergere
% set toto "il etait une $a"
% puts $toto
il etait une bergere
% set toto {il etait une $a}
% puts $toto
il etait une $a
```

Les guillemets permettent l'utilisation de caractères blancs dans une chaîne, les accolades aussi, mais celles-ci interdisent l'évaluation des variables éventuelles contenues dans la chaîne.

Il est aussi possible d'utiliser des espaces dans une chaîne de caractères en les faisant précéder d'un anti-slash. Celui-ci permet de considérer le caractère qui le suit comme un caractère normal.

*% **set** toto il\ etait\ une\ $a*
*% **puts** $toto*
il etait une bergere
Ainsi si une chaîne de caractères doit contenir des guillemets ou des accolades:
*% **set** toto \{il\ etait\"\ une\nbergere* # \n force un retour chariot
*% **puts** $toto*
{il etait" une
bergere

Remarque: la concaténation de chaînes peut se faire de manière automatique:
*% **set** Aut "un aut"*
*% **set** Re re*
*% **set** toto "un train peut en cacher AutRe"*
un train peut en cacher un autre

2. Listes:

Tcl/Tk permet de tout considérer comme une liste, par exemple:
*% **set** toto "il etait une bergere"*
*% **llength** $toto* #fonction retournant le nombre d'éléments de la #liste en argument
4

Il est possible de construire explicitement des listes par l'utilisation du constructeur *list* ou par l'utilisation d'accolades:
*% **set** toto {il {etait une bergere}}* #liste contenant deux éléments
*% **puts** $toto*
il {etait une bergere}
*% **llength** $toto* #nombre d'éléments de la liste
2
*% **llength** [**lindex** $toto 1]* #nombre d'éléments du deuxième élément de la #liste: celles-ci sont indexées à partir de 0
3
*% **set** toto [**list** il [**list** etait une bergere]]* #donne le même résultat qu'en utilisant les #accolades. Cependant, permet l'utilisation #de variables dans la fabrication de la liste, # ce que les accolades empêchent.

La fonction la plus intéressante pour le parcours de liste est *foreach*, qui permet d'effectuer un ensemble de commandes sur (ou a partir de) l'ensemble des éléments d'une liste:

*%**set** maliste "Il etait une bergere"*
*%**foreach** mot $maliste {*
* **puts** $mot*
}
Il
etait
une

bergere

La propriété intéressante de *foreach* est qu'elle permet de traiter les éléments de l'ensemble par couple, triplet, etc...

```
%foreach {motimpair motpair} $maliste {
 puts $motpair                  #seul un mot sur deux est affiché
}
etait
bergere
```

3. Entiers et flottants:

Comme Tcl/Tk considère que tout est chaîne de caractères, il est nécessaire de lui spécifier qu'une opération arithmétique est désirée:

```
% set a 23+12
23+12
% set a [expr 23+12]        #utilisation de la fonction expr pour des calculs arithmétiques
35
```

La différence entre entiers et flottants se fait de manière implicite suivant le type des opérandes:

```
% expr 23/3                 #calcul entier car les deux arguments sont des entiers
7
% expr 23.0/3               #un des arguments est un réel => calcul flottant
7.66667
```

4. Tableaux:

Les tableaux sont des variables possédant des entrées:

```
% set tableau(oncle) Tom          #l'entrée oncle du tableau reçoit la chaîne "Tom"
% puts $tableau(oncle)            #référence au contenu d'une variable de type tableau
Tom
% set i oncle
% puts $tableau($i)               #encore une référence, mais cette fois ci  par une
                                  #variable
Tom
```

On peut aussi créer des tableaux à plusieurs dimensions, dans ce cas les noms d'entrées sont séparés par des virgules :

```
% set tab2(case,enpaille) oncle      #l'entrée référencée par case et enpaille reçoit
                                     #"oncle"
% puts $tableau($tab2(case,enpaille)) #affichage de l'entrée oncle de tableau
```

Tom

Limitation: une variable de type tableau ne peut pas être créée dans une variable autre et vice-versa:

```
% set toto 12          #toto est donc connu par l'interpréteur comme une variable chaîne
% set toto(a) 10       #assignation incorrecte
can't set "toto(a)": variable isn't array
% unset toto           #le contenu de la variable est libéré, de plus la variable toto est "oubliée"
% set toto(a) 10       #assignation maintenant correcte car toto est une nouvelle variable
```

B. Structures de programme

1. Conditionnelles:

Une conditionnelle s'écrit sous la forme:

if condition bloc
ou bien
if condition bloc1 **else** bloc2
ou bien
if condition bloc1 **elseif** condition2 bloc2 ... **else** bloc n

Remarque importante: cette instruction s'écrit sur la même ligne, on ne peut insérer des sauts de lignes qu'à l'intérieur des accolades, ainsi:

```
% set a 4
% if {$a > 0} {
puts $a}                #incorrect car le else n'est pas sur la même ligne que le bloc
else {puts -$a}
```

Conseil: toujours écrire les tests sous la forme suivante:

```
if {condition} {
  bloc
} else {
  bloc
}
```

La condition doit, contrairement au C, être un "vrai" booléen (0 pour faux ou 1 pour vrai). Si la condition est composée de plus d'une opération arithmétique, il faut la mettre sous la forme *[expr* condition*]* qui force une évaluation mathématique et/ou booléenne de la condition.


Attention, les opérations mathématiques sont interdites sur les chaînes de caractères, en particulier il faut remplacer:
$chaîne1 == $chaîne2
par

[***string compare*** *$chaîne1 $chaîne2]==0*.
string compare est une fonction qui renvoie 0 si les deux chaînes comparées sont identiques. La bibliothèque ***string*** regroupe un nombre appréciable de commandes de traitement de chaînes de caractères.

2. Boucle "pour":

Un boucle "pour" s'écrit sous la forme:

for initialisation condition incrémentation bloc

De la même manière que les tests, une boucle "pour" s'écrit sur une même ligne, il ne peut y avoir des sauts de ligne qu'à l'intérieur d'accolades.

Exemple d'utilisation de la boucle pour:

```
% for {set i 1} {$i<=10} {incr i} {
  puts $i
}
```

Conseil d'écriture de la boucle pour:

```
for {initialisations} {condition} {incrémentations} {
  bloc
}
```

3. Boucle "tant que":

Une boucle "tant que" s'écrit sous la forme:

while condition bloc

Comme toute instruction Tcl/Tk, cette commande s'écrit sur la même ligne. Ainsi on ne peut mettre des sauts de lignes que dans des blocs délimités par des accolades:

Exemple d'utilisation:

```
% set a 1
% while {$a>0} {
puts $a
incr a -1        #décrémentation de a
}
```

Conseil d'écriture d'une boucle "tant que":

```
while {condition} {
  bloc
}
```

4. Branchement à choix multiples:

Le branchement à choix multiples s'écrit de la manière suivante:

switch chaîne modèle$_1$ bloc$_1$ ... modèle$_n$ bloc$_n$
ou bien
switch chaîne modèle$_1$ bloc$_1$ ... modèle$_n$ bloc$_n$ **default** blocdefault

avec blocdefault qui est exécuté si aucun modèle ne correspond à la chaîne.

Exemple:

```
% switch $a {
  toto {puts "c' est toto"}
  titi {puts "c'est titi"}
 default {puts "je ne le connais pas"}
}
```

Conseil d'écriture:

```
switch chaîne {
  modèle1 {
    bloc1
  }
  modèle2 {
    bloc2
  }
  ...
}
```

 les modèles doivent être des constantes (i.e pas de $toto pour modèle).

C. Fonctions

Les fonctions permettent d'introduire une modularité (relativement faible) dans un programme Tcl/Tk. La définition d'une fonction se fait de la façon suivante:
proc nom paramètres corps

En fait elle s'écrira le plus souvent:

```
proc nom_proc {param1 param2 ... paramn} {
  bloc
}
```

Les fontions sont toutes censées renvoyer une valeur (qui peut être une chaîne vide). Dans le cas où une instruction **return** est rencontrée lors de l'évaluation de la fonction, son évaluation est stoppée et la valeur suivant le **return** est renvoyée. Dans le cas où il n'y a aucune instruction **return**, le résultat de la dernière commande exécutée par la fonction est renvoyé par celle-ci.

La valeur d'une fonction est retournée lorsque celle-ci est appelée. L'appel d'une fonction se fait soit de manière procédurale (on ne conserve pas le résultat):

```
% nom_proc   param1 param2 ... param3
#le résultat de la fonction nom_proc est ignoré bien que la fonction soit interprétée
```

Soit l'appel se fait par la mise entre crochets de l'appel:

```
% set a [nom_proc param1 param2 ... paramn]
#la variable a contiendra le résultat retourné par la fonction appelée
```

Exemple:

```
% proc racines {a b c} {
  #calcule les racines du polynôme ax2+bx+c
  set delta [expr pow($b,2) - (4*$a*$c)]
   if {$delta==0} {
    return [expr -$b/(2.0*$a)]
  } elseif {$delta>0} {
    return [list [expr (-$b+sqrt($delta)) / ($a*2.0)] [expr (-$b-sqrt($delta)) /($a* 2.0)]]
  } else {
    return ""
  }
}
% puts [racines 1 2 1]
-1.0
% puts [racines 2 -5 2]
2.0 0.5
```

1. Utilisation des variables dans les procédures:

Toutes les variables affectées à l'extérieur des fonction sont considérées comme globales. Ainsi l'accès en lecture ou en écriture à une variable à l'intérieur d'une fonction se fait sur une variable locale par défaut. Pour accéder à une variable globale (par exemple *toto*) à l'intérieur d'une fonction, il faut déclarer celle-ci comme globale à l'intérieur de la fonction (de préférence avant tout accès à cette variable) avec le mot clé **global** (par exemple *global toto*).

En fait, Tcl/Tk possède une pile de variables. Au niveau le plus haut (hors de toute fonction), se trouvent les variables globales (couche 0). Lorsque l'on se trouve à l'intérieur d'une fonction appelée depuis la couche zéro, les variables déclarées sont empilées dans la couche 1, puis si il y a appel récursif, sur le niveau 2 etc... Ainsi le mot clé **global** fait un lien entre la variable locale et la variable globale de même nom. Il est aussi possible à l'aide de **upvar** de lier une variable locale à une variable de niveau relativement plus élevé ou d'un niveau absolu.

Tous les paramètres des fonctions étant passés par valeur, ils sont en IN uniquement. Il existe cependant un moyen de passer des paramètres par nom, donc en IN OUT, ainsi que des tableaux qui ne peuvent pas être passés par valeur, mais uniquement par nom.

```
% proc AfficherValeursTableauEtDetruire {tableau} {
  upvar $tableau T    #la variable locale T est liée à la variable tableau du niveau appelant
  puts [array get T]   # array get renvoie la liste des couples {Nom_d_entrée Valeur}
  unset T
}
% set toto(i) 1
% AfficherValeursTableauEtDetruire toto
i 1
% AfficherValeursTableauEtDetruire toto   #toto a été détruit et n'existe plus
can't unset "T": no such variable
```

Il est à remarquer que les variables passées en IN sont passées par valeur ($nomvariable) alors que les variables passées en IN OUT grâce à **upvar** sont passées par nom (par exemple *incr i* et non pas *incr $i* à moins bien sûr que *i* contienne un nom de variable).

D. Quelques mots sur l'existence des variables

Les variables sont soit des variables de types scalaires (listes, chaînes, entiers, réels) soit des variables de type tableau. Ainsi l'accès à une variable de type tableau peut provoquer une erreur si on essaie d'accéder au tableau tout entier, et l'accès à une variable scalaire peut provoquer une erreur si on essaie d'accéder à celle-ci comme à un tableau.

Pour savoir si une variable est de type tableau, il suffit de vérifier que *array exists nomvariable* renvoie vrai (1).

De plus la cause la plus fréquente d'erreurs est la non existence de variables que l'on tente de lire ou de supprimer. Pour savoir si une variable existe, il suffit de vérifier que *info exists nomvariable* renvoie vrai.

E. Un traitement puissant des exceptions

Tcl n'a rien à envier aux langages évolués proposant des méthodes de traitement des exceptions, tels Ada ou même Java. En effet Tcl permet par le biais de la fonction **catch** de récupérer toute erreur, et par les options de la commande **return** de lever des exceptions agrémentées de texte. Dans le cas normal, une fonction retourne une valeur donnée par **return** *une_valeur* qui rend la main au bloc ayant appelé la fonction. Son code de terminaison est TCL_OK dans ce cas, et l'interpréteur continue l'exécution de façon normale. Il est possible de modifier le comportement de l'interpréteur en modifiant le code de terminaison d'une fonction par *return -code erreur une_chaîne_décrivant_l'erreur*. De cette manière, l'exception TCL_ERROR remonte la pile d'appel jusqu'au premier bloc appelant ayant prévu de traiter les exceptions. Si aucun bloc appelant ne contenait de **catch**, c'est interpréteur lui-même qui traite l'erreur. Dans le cas de l'interpréteur *wish*, une fenêtre est affichée contenant le texte de *une_chaîne_décrivant_l'erreur*, un bouton ok, un bouton *skip* message, et un bouton *stack trace* permettant de visualiser la pile d'appel (en source Tcl) que l'exception a remontée. Par contre, si l'exception rencontre un **catch**, celle-ci n'est plus levée et la fonction catch renvoie un résultat non nul.

Par exemple si l'on veut écrire une fonction ouvrant un fichier en lecture, et en lisant 1 chiffre sur la première ligne qui indique le nombre de lignes du fichier à afficher, il est très

laborieux d'écrire une fonction qui testera si le fichier existe, si le programme a les privilèges suffisants pour le lire, si la première ligne existe, si elle contient un chiffre, si le fichier contient un nombre de ligne supérieur au chiffre lu en premier. Ce cas est typique d'une fonction contenant un traite-exceptions:

```
proc AfficheContenu {NomFichier} {
 if {[catch {                        #Récupération d'une éventuelle erreur
  set F [open $NomFichier r]         # open sert à ouvrir un fichier et renvoie son descripteur
  gets $F NbLigneALire               # gets lit dans un fichier
  for {set i 1} {$i<=$NbLigneALire} {incr i 1} {
   gets $F ligne
   puts $ligne
   }
  close $F
} Erreurs]!=0} {                     #Si catch renvoie un résultat non nul il y a une erreur
 puts $Erreurs                       #dont le texte explicatif est dans Erreurs
}
```

Il est également possible de définir ses propres exceptions. Prenons l'exemple d'un petit interpréteur, sensible aux majuscules mais « intelligent », écrit en Tcl, qui pourra traiter un fichier mot par mot, et réagir suivant la syntaxe et la sémantique définie pour le langage. On écrira sans doute une fonction qui prend un mot en argument, change l'état de l'automate du langage en fonction du contexte et exécute des commandes Tcl avant de rendre la main à la fonction appelante qui lui passera le mot suivant, et ainsi de suite jusqu'à trouver une erreur ou jusqu'à la fin du fichier à traiter. Cet interpréteur possédera une fenêtre dans laquelle seront affichés les éventuels avertissements et erreurs détectés durant l'interprétation du fichier mais qui laissera remonter les erreurs liées aux commandes appelées par l'utilisateur.
Voici à quoi pourra ressembler la fonction traitant les entrées mot par mot:

```
proc traite_mot {mot} {
 global EtatActuel Automate Actions
 if {[info exists Automate($EtatActuel,$mot)]} {     #Le mot était attendu dans le contexte
  set EtatActuel $Automate($EtatActuel,$mot)         #Changement d'état
  eval $Actions($EtatActuel)                         #Actions associées
 } else {
  if {[info exists Automate($EtatActuel,[string tolower $mot])]} {
   set EtatActuel $Automate($EtatActuel, [string tolower $mot])
   #Il n'y a peut-être qu'une erreur de casse
   eval $Actions($EtatActuel)                        #Actions associées
   return -code 1 "Assuming $mot is [string tolower $mot]"
   #on lève une exception qui sera considérée comme un warning
 } else {
  return -code 2 "Unexpected $mot"
  #Sinon on lève une exception qui sera considérée comme une erreur
 }
}
```

La fonction appelante passera chaque mot du fichier à la fonction *traite_mot* en ne récupérant que les exceptions de type 1 et 2 qu'elle considérera respectivement comme des warnings et des erreurs.

```
proc Interprete {NomFic} {
  ...
#Création d'une fenêtre de texte dans laquelle seront affichés les éventuels warnings et erreurs
 set F [open $NomFic r]
 while {[eof $F]==0} {                          #Tant qu'il reste des lignes à lire
  gets $F ligne
  foreach mot $ligne {
   set Resultat [catch {traite_mot $mot} Erreurs]
   switch $Resultat {
    1 {#Affichage de Warning:$Erreurs dans la fenêtre de texte}
    2 {
      #Affichage de Erreur:$Erreurs dans la fenêtre de texte
     close $F
      return 1
    }
    default {
      close $F
      return -code $Resultat $Erreurs
    }
    #Les autres exceptions ne sont pas stoppées à ce niveau
   }
  }
 }
 close $F
}
```

La fonction *interpreter* pourra elle-même être appelée à l'intérieur d'un bloc *catch*...

F. Redéfinition des fonctions

Il existe une autre méthode de récupérer les erreurs, non pas pour faire un traitement spécial suivant l'erreur levée comme pour les exceptions, mais tout simplement pour éviter qu'une erreur non récupérée puisse s'afficher et être vue par l'utilisateur. En effet Tcl/Tk appelle une procédure nommée ***error*** ou bien ***bgerror*** lorsqu'une erreur arrive au niveau de l'interpréteur. Ceci se traduit par l'apparition d'une boîte de dialogue contenant la trace du programme lorsque l'on se trouve sous wish. Cela peut être gênant lorsque l'application est développée, d'autant plus que la plupart des erreurs qui peuvent advenir lorsque le programme est débuggé sont des erreurs d'interface : elles sont souvent dûes à des erreurs internes de déroulement de menu.

Pour éviter ce genre de désagrément, il suffit de créer une nouvelle procédure ***error*** et une nouvelle procédure ***bgerror*** qui ne feront rien ou bien qui placeront l'erreur dans un fichier quelconque. C'est un des avantages des langages interprétés.

Cependant, il peut être intéressant de conserver le code de ces fonctions et de pouvoir les appeler dans certains cas. Pour cela, il suffit d'utiliser la commande ***rename*** :

```
rename error error2
rename bgerror bgerror2
proc error {args} {
  puts "Je fais ce que je veux avec les erreurs"
  error2 $args
}
proc bgerror {args} {
  puts "Je fais ce que je veux avec les erreurs en background"
}
```

Voilà posées les bases de la programmation Tcl. Voyons maintenant comment programmer en utilisant les fonctionnalités de Tk.

III. Programmation Tk

Tk (*Tool Kit*) est une surcouche graphique de Tcl, ou une "bibliothèque graphique" de haut niveau. Chaque objet graphique est un Widget: il a un unique père et peut avoir une descendance. Une fenêtre graphique contenant des objets est donc un arbre, la racine étant la fenêtre principale (créée automatiquement par *wish*) et toujours nommée ".".
Un widget est toujours créé de la même manière:
nomwidget .ancètre1.ancètre2....ancètren.moi options_de_création_du_widget
Par exemple si l'on veut créer un bouton dans la fenêtre principale, la commande est:
***button** .monbouton **-text** {Bonjour le monde} **-command** {**puts** "Salut mon bouton"}*

Il faut maintenant faire apparaître le bouton dans la fenêtre. Pour cela on utilise soit le *placing* qui consiste à placer les éléments de manière statique (en x=32 et y=12) soit le *packing* qui consiste à placer les *widgets* les uns par rapport aux autres, soit le *gridding*. La deuxième méthode permet plus de souplesse.

Une fois créé, un widget possède un nom donné lors de sa création, dans l'exemple ci-dessus, le nom du widget est *.monbouton*, et on pourra le détruire par la commande *destroy .monbouton*, ou bien modifier son texte et son comportement par:
*.monbouton **configure** -text {Au revoir le monde} **-command** {**puts** "Au revoir le monde";**destroy** .monbutton}*
Dans ce cas, le texte affiché change, et lorsque le bouton est actionné, "Au revoir le monde" est affiché, puis le bouton est détruit.


Tcl/Tk ne met à jour l'affichage que lorsqu'il n'a plus rien à faire. Pour l'obliger à mettre à jour l'affichage alors qu'il a des commandes à traiter, il faut utiliser la commande ***update***.

Pour commencer, voyons quels widgets Tk nous propose:

A. Hiérarchiser pour mieux contrôler ses widgets

Le meilleur moyen de connaître les widgets est de lire l'aide Tcl/Tk: pour chacun une foule d'options sont disponibles. Nous donnons juste ici leur liste avec une brève description de leurs fonctionnalités les plus usuelles.

1. Les *widgets container* : la frame et la fenêtre (*toplevel*)

Les fenêtres de Tk sont hiérarchisées. Ainsi un *widget*, lors de sa création, possède des ancêtres, chacun étant séparé par un point de son père: $ancêtre_1.ancêtre_2....widget$. Ainsi certains *widget* peuvent en contenir d'autres, comme les fenêtres (*toplevel*), ou les frames (*frame*). Les *widgets text* et *canvas* font aussi partie de cette catégorie mais ne permettent que de placer explicitement leurs *widgets* fils en donnant leur position absolue. Les *toplevel* et les *frame*, quant à elles, permettent de placer leurs fils de manière relative les uns par rapport aux autres par la technique du *packing*.

Lorsqu'une application Tcl/Tk démarre, une *toplevel* est automatiquement créée, c'est la *toplevel* ".". Elle est l'ancêtre de tous les *widgets* qui seront créés par l'application, et sa destruction engendre la fin de l'application.

On peut à loisir créer d'autres fenêtres par la commande *toplevel*, en voici un exemple:

toplevel *.top1*
#Création d'une fenêtre de nom *.top1*
#Certaines options sont disponibles sur les toplevel, mais le mieux est d'utiliser la
 # commande wm
wm maxsize *.top1 1024 768*
#Taille maximale de la fenêtre
wm minsize *.top1 400 200*
#Taille minimale de la fenêtre
wm positionfrom *.top1* ***user***
#Pour les systèmes d'exploitation qui demandent à l'utilisateur de placer manuellement la
fenêtre. Conseil: écrire cette instruction à chaque fois
wm sizefrom *.top1* ***user***
#Pour les systèmes d'exploitation qui demandent à l'utilisateur de dimensionner manuellement
la fenêtre
#Conseil: écrire cette instruction à chaque fois
wm title *.top1 {Une fenêtre}*
#Titre de la fenêtre
wm geometry *.top1 500***x***300+10+10*
La fenêtre aura une taille de 500x300 pixels et sera positionnée en 10,10 sur l'écran
wm protocol *.top1 WM_DELETE_WINDOW {puts "Je ferme le fenêtre";destroy .top1}*
Lorsque l'utilisateur ferme la fenêtre, par exemple en cliquant sur la croix en haut à droite
sous windows95, l'application écrit *Je ferme la fenêtre* avant de la fermer.
C'est une commande très utile, notamment pour demander à l'utilisateur lorsqu'il ferme la
fenêtre principale: "Êtes-vous sûr que vous voulez vraiment quitter mon beau programme
pour de vrai sûr de sûr ...?"

Après avoir créé une fenêtre *toplevel*, la première chose à faire est de créer une frame à l'intérieur de la fenêtre. Une frame est un *widget* le plus souvent invisible servant à hiérarchiser les *widgets* situés dans une fenêtre. Il peut cependant avoir une couleur, un bord, etc...

Options les plus utilisées pour les frames	
-background *couleur*	La *couleur* spécifiée est utilisée pour afficher la frame
-borderwidth *épaisseur*	La frame possède une épaisseur 3D de *épaisseur* pixels
-relief *r*	L'épaisseur 3D donne à la frame l'allure 3D d'épaisseur donnée par l'option *-borderwidth* spécifiée par *r* : *r* = **raised**  , *r* = **sunken**  , *r* = **flat**  , *r* = **ridge**  , *r* = **solid**  , *r* = **groove**  .
Commandes les plus utilisées sur les frames	
cget *option*	Renvoie la valeur de l'option spécifiée de la frame
configure *?options?*	Change les options spécifiées (voir plus haut) de la frame

Voici comment nous créons la première frame de notre fenêtre:
frame *.top1.f*
Création d'une frame fille de *.top1*

pack configure *.top1.f* ***-expand*** *1* ***-fill both***
La frame est affichée à l'intérieur de la fenêtre, l'option *-expand 1* force la frame à réserver le
plus de place possible dans la fenêtre et l'option *-fill both* la force à occuper toute la place
qu'elle a réservée dans les deux sens (en x et en y)

 Un *widget* est invisible s'il n'est pas *packé*, *placé*, ou *griddé*.

La commande permettant de *packer* un widget est de la forme :

pack configure *widget ?options?*

Les options les plus couramment utilisées sont données dans le tableau ci-dessous:

Option de la commande *pack configure*	
-after *autrewidget*	Par défaut la commande *pack configure* donne la priorité aux *widgets* dans l'ordre d'arrivée. Ainsi si deux widgets demandent à être en haut d'une frame, le premier à le demander sera au-dessus de l'autre. Cette option permet d'insérer un widget dans l'ordre d'arrivée.
-before *autrewidget*	Similaire à l'option *-after*, sauf qu'il place le widget juste avant le *widget autrewidget*.
-expand *0_ou_1*	Si *-expand* est mis à vrai (1), le *widget* consommera tout l'espace possible (partagé équitablement avec les autres *widgets* ayant été *packés* avec la même option.
-fill *style*	Dans le cas où l'espace obtenu est plus grand que la taille demandée par le *widget*, celui-ci sera agrandi pour remplir l'espace horizontalement (*style* = **x**), verticalement (*style* = **y**) ou bien complètement (*style* = **both**).
-ipadx *pixels*	Spécifie un bord de largeur *pixels* à rajouter à gauche et à droite à l'intérieur du *widget* en plus de sa taille demandée.
-ipady *pixels*	Spécifie un bord de hauteur *pixels* à rajouter en haut et en bas à l'intérieur du *widget* en plus de sa taille demandée.
-padx *pixels*	Spécifie un bord de largeur *pixels* à rajouter à gauche et à droite à l'extérieur du *widget* en plus de sa taille demandée.
-pady *pixels*	Spécifie un bord de hauteur *pixels* à rajouter en haut et en bas à l'extérieur du *widget* en plus de sa taille demandée.
-side *côté*	C'est l'option principale de cette commande, puisque la valeur de *côté* (**top**, **bottom**, **left**, **right**) dit ou placer le *widget* dans son père.

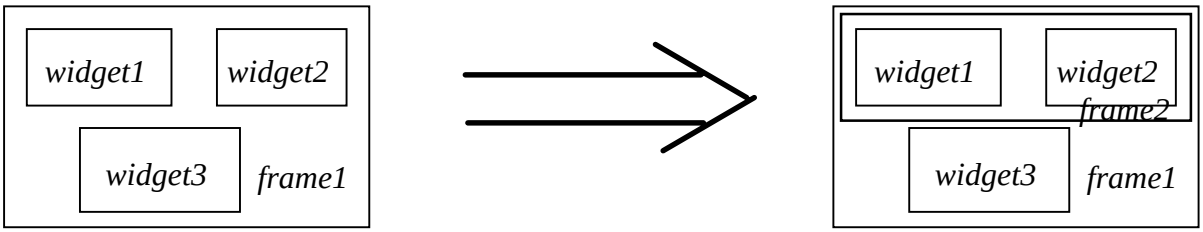
On peut masquer un *widget* par la commande ***pack forget*** *nom_widget*. Ainsi le *widget* n'est plus affiché jusqu'à ce qu'il soit de nouveau packé.
Maintenant créons deux éléments dans la fenêtre *.top1*: un bouton, en bas, et un message qui devra prendre le plus de place possible à l'intérieur de la fenêtre. On peut le faire ainsi:

button *.top1.f.un_bouton* ***-text*** *{Ok}* ***-command*** *{destroy .top1}*
#Création d'un bouton fils de *.top1.f* contenant le texte "Ok" et fermant le fenêtre *.top1*
pack configure *.top1.f.un_bouton* ***-side bottom***
Met le bouton en bas de la frame (donc de la fenêtre)
message *.top1.f.mesg* ***-text*** *{Clickez sur ok pour fermer\n cette fenêtre!!!}*

#Création d'un message
pack configure *.top1.f.mesg* ***-side top -expand*** *1* ***-fill both***
Affiche le message en haut de la frame en lui allouant le plus d'espace possible dans
la fenêtre

2. Hiérarchie de Tk

En étudiant la commande ***pack configure***, et notamment son option ***-side***, il est clair que les *widgets* peuvent être facilement *packés* les uns par rapport aux autres soit horizontalement (valeurs ***left*** et ***right*** pour ***-side***), soit verticalement (valeurs ***top*** et ***bottom*** pour ***-side***). *Packer* trois *widgets* en en mettant un en haut à gauche, l'autre en haut à droite, et le dernier en dessous des deux autres n'est pas naturel si ces trois *widgets* sont dans la même frame. La solution est de hiérarchiser la frame en une sous frame, qui contiendra les deux widgets du haut *packés* horizontalement, et le troisième *widget* qui sera frère de la sous frame créée, et qui sera *packé* verticalement par rapport à celle-ci.

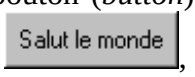


B. Les différents widgets

Dans ce sous chapitre, nous considérerons qu'il existe une fenêtre *.top1* contenant une frame *.top1.f*, qui est *packée*, existe. Pour les détails de création de ces éléments, se référer au sous-chapitre précédent.

1. Les boutons

a) Le bouton

Le bouton (*button*) est le widget le plus simple. On lui associe un texte statique ou dynamique 

, comme le contenu d'une variable globale, qui sera affiché sur le bouton et pourra être modifié soit explicitement par l'accès au texte du bouton, soit implicitement par la modification de la variable associée au texte du bouton si on lui avait associé une variable. Un bouton peut aussi afficher une image  . En général on lui associera une commande à appeler lorsque l'utilisateur l'actionne.
Exemples de créations et de manipulations de boutons:

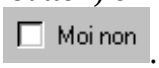
button *.top1.f.b* ***-text*** *"Salut le monde"*

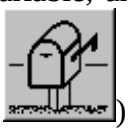
pack configure *.top1.f.b* ***-side top***
#Création et affichage d'un bouton, son texte, statique, est Salut le monde.
.b ***configure -state disabled***
#Le bouton n'est plus disponible
button *.top1.f.b2* ***-textvariable*** *le_texte_du_bouton*
pack configure *.top1.f.b2*
#Création et affichage d'un bouton dont le texte est le texte contenu dans la variable
le_texte_du_bouton, dont toute modification affecte le texte du bouton
image create photo *ouvrirfichier* ***-format gif -file*** */images/ouvrir.gif*
Création d'une image à partir d'une image gif existante, cette image est
nommée *ouvrirfichier*
button *.top1.f.b3* ***-image*** *ouvrirfichier*
pack configure *.top1.f.b3* ***-side left***
Création et affichage d'un bouton dans la fenêtre principale son image est ouvrirfichier

Options les plus utilisées pour les boutons	
-command *com*	La commande *com* spécifiée sera appelée lorsque le bouton sera actionné
-image *img*	L'image *img* spécifiée est affichée sur le bouton
-state *s*	Le bouton est disponible (s = ***normal***) ou indisponible (s = ***disabled***)
-text *t*	*t* est le texte associé au bouton
-textvariable *v*	*v* est une variable globale dont le contenu est affiché sur le bouton, et dont toute modification entraîne la modification du texte du bouton
Commandes les plus utilisées sur les boutons	
cget *option*	Renvoie la valeur de l'option spécifiée du bouton.
configure *?options?*	Change les options spécifiées (voir plus haut) du bouton

b) La boîte à cocher

La boite à cocher (*checkbutton*) diffère du bouton normal car elle a deux états: cochée 

et non cochée 

. On pourra comme pour le bouton lui associer un texte, le contenu d'une variable, une image ou même deux (une lorsqu'il est coché 
, une lorsqu'il ne l'est pas 
). On pourra même associer une commande aux actions de l'utilisateur sur cette boite.
Exemples de création de boîtes à cocher:

checkbutton *.top1.f.c1* ***-text*** *"Je suis coché"* ***-variable*** *EtatC1*
pack configure *.top1.f.c1* ***-side left***
#Création d'une boîte à cocher. Son état est dans la variable (globale) *EtatC1* (=1 si coché, 0
sinon)
set *EtatC1 1*
#*.top1.f.c1* est coché
image create bitmap *FlagUp* ***-file*** */images/flagup.bmp* ***-maskfile*** */images/flagup.bmp*

image create bitmap *FlagDown* ***-file*** */images/flagdown.bmp* ***-maskfile*** */images/flagdown.bmp*
#Création de deux images à base de bitmaps au format X11 bitmap
checkbutton *.top1.f.c2* ***-image*** *FlagDown* ***-selectimage*** *FlagUp* ***-variable*** *EtatC2* ***-indicatoron*** *0*
#Création d'une boîte à cocher. Son état est dans la variable (globale) *EtatC2*
Il ne contient pas d'indicateur (case cochée ou non) car son état sera traduit par le bitmap
affiché (*FlagUp* lorsqu'elle est cochée, *FlagDown* sinon)

Options les plus utilisées pour les boîtes à cocher	
-command *com*	La commande *com* spécifiée sera appelée lorsque la boîte sera cochée ou décochée, dans ce cas, la variable associée à la boite (option *-variable*) contient le nouvel état de la boîte
-image *img*	L'image *img* spécifiée est affichée sur la boîte
-indicatoron *0_ou_1*	La boîte affiche une petite boîte qui contiendra une coche si la boîte est cochée
-selectimage *img*	Lorsque la boîte est cochée, l'image *img* spécifiée est affichée dans la boîte
-state *s*	La boîte est disponible (s = ***normal***) ou indisponible (s = ***disabled***)
-text *t*	Le texte *t* est associé à la boîte
-textvariable *v*	*v* est une variable globale dont le contenu est affiché dans la boîte et dont toute modification entraîne la modification du texte de la boîte
-variable *v*	*v* est une variable globale associée à l'état de la boîte, elle contient 1 si la boîte est cochée, 0 sinon, et réciproquement coche ou décoche la boîte lorsqu'elle est modifiée.
Commandes les plus utilisées sur les boîtes à cocher	
cget *option*	Renvoie la valeur de l'option spécifiée de la boîte
configure *?options?*	Change les options spécifiées (voir plus haut) de la boîte

c) Le bouton radio

Le bouton radio (*radiobutton*) 

est en fait une extension de la boîte à cocher: un ensemble de boutons radio utilisant la même variable sont en exclusion mutuelle (i.e. quand un ensemble de boutons radio partagent la même variable (option *-variable*), seuls ceux qui ont la même valeur (option *-value*) peuvent être sélectionnés en même temps).
Exemples de boutons radio:

radiobutton *.top1.f.r1* ***-text*** *"Selection de groupe 1"* ***-variable*** *UneVariable* ***-value*** *1*
pack configure *.top1.f.r1* ***-side top***
Création d'un bouton radio
set *UnTexte "Selection de groupe 2"*
radiobutton *.top1.f.r2* ***-textvariable*** *UnTexte* ***-variable*** *UneVariable* ***-value*** *2*
pack configure *.top1.f.r2* ***-side top***
Un bouton radio en exclusion de *.top1.f.r1*
radiobutton *.top1.f.r3* ***-text*** *"Autre sélection de groupe 1"* ***-variable*** *UneVariable* ***-value*** *1*
pack configure *.top1.f.r3* ***-side top***
Autre bouton radio qui sera sélectionné si et seulement si *.top1.f.r1* est sélectionné car il a la même
valeur (*-value 1*) que lui.

Remarque: un bouton radio peut, comme les boîtes à cocher, être associé à des images.

Options les plus utilisées pour les boutons radio	
-command *com*	La commande *com* spécifiée sera appelée lorsque le bouton sera sélectionné ou désélectionné, dans ce cas, la variable associée au bouton (option *-variable*) contient le nouvel état du bouton
-image *img*	L'image *img* spécifiée est affichée sur le bouton
-indicatoron *b*	Si *b* = 1, le bouton affiche une petite boîte qui contiendra une coche si le bouton est cochée
-selectimage *img*	Lorsque le bouton est sélectionné, l'image *img* spécifiée est affichée dans le bouton
-state *s*	Le bouton est disponible (s = **normal**) ou indisponible (s = **disabled**)
-text *t*	Texte associé au bouton
-textvariable *v*	*v* est une variable globale dont le contenu est affiché dans le bouton
-variable *v*	*v* est une variable globale associée à l'état du bouton, elle contient 1 si le bouton est sélectionné, 0 sinon, et réciproquement sélectionne ou désélectionne le bouton lorsqu'elle est modifiée.
Commandes les plus utilisées sur les boutons radio	
cget *option*	Renvoie la valeur de l'option spécifiée du bouton radio
configure *?options ?*	Change les options spécifiées (voir plus haut) du bouton radio

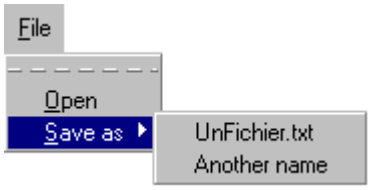
d) Le bouton de menu et le menu déroulant

Un bouton de menu (*menubutton*) n'est ni plus ni moins qu'un bouton que l'on associe à un menu déroulant. Lorsque celui-ci est actionné, le menu associé par l'option *-menu* est déroulé.

Un menu (*menu*) est un widget invoqué (en fait déroulé) par la commande *tk_popup*. Il est possible de l'associer à un bouton de menu (menu déroulant) ou à un événement utilisateur (par exemple click droit) par le *binding* d'un click sur le bouton droit à la commande *tk_popup*.
En exemple voici la création d'un menu déroulant:

menubutton *.top1.f.file* ***-text*** *File* ***-menu*** *.file.menu* ***-underline*** *0*
Création d'un bouton de menu. L'option *underline* souligne la première lettre (n°0) du
mot File et un raccourci clavier (Alt+f) est associé au bouton de menu.
menu *.top1.f.file.menu*
Création d'un menu fils de *.top1.f.file*
.top1.f.file.menu ***add command -label*** *Open* ***-command*** *Ouvrir* ***-underline*** *0*
La commande nommée Open, qui appelera la fonction Ouvrir quand elle sera sélectionnée
est ajoutée au menu *.top1.f.file.menu*, l'option *underline* va souligner la lettre n°0 (1ère lettre)
du texte affiché dans le menu (Open) et un raccourci clavier est créé.
.top1.f.file.menu ***add cascade -label*** *"Save as"* ***-menu*** *.file.menu.save* ***-underline*** *0*
La deuxième ligne du menu est une cascade (affiche un autre menu quand il est sélectionné)
menu *.top1.f.file.menu.save* ***-tearoff*** *0*
Création d'un menu
.top1.f.file.menu.save ***add command -label*** *$FileName* ***-command*** *Sauver* ***-underline*** *-1*

Ajout d'une commande dans le menu, en supposant que FileName est la variable
contenant le nom du fichier actuel. Aucun raccourci clavier n'est associé (*underline -1*).
.top1.f.file.menu.save ***add command -label*** *"Another name"* ***-command*** *SauverSous*
Ajout d'une commande.



Options les plus utilisées pour les boutons de menu	
-image *img*	L'image *img* spécifiée est affichée sur le bouton
-menu *m*	Lorsque le bouton est sélectionné, le menu *m* spécifié est déroulé
-state *s*	Le bouton est disponible (*s* = ***normal***) ou indisponible (*s* = ***disabled***)
-text *t*	Texte associé au bouton
-textvariable *v*	*v* est une variable globale dont le contenu est affiché dans le bouton
-underline *u*	Si *u* ≥0, la lettre correspondante dans le texte du bouton (la première lettre étant la numéro 0) est soulignée, et le bouton est actionné lorsque l'utilisateur appuie sur Alt+la_lettre_correspondante
Commandes les plus utilisées sur les boutons de menu	
cget *option*	Renvoie la valeur de l'option spécifiée du bouton de menu
configure *?options ?*	Change les options spécifiées (voir plus haut) du bouton de menu

Options les plus utilisées pour les menus	
-postcommand *com*	Spécifie une commande appelée à chaque fois que le menu est déroulé
-tearoff *b*	Si *b* = 1, la première ligne du menu est une ligne pointillée qui, lorsqu'elle est actionnée, crée une fenêtre contenant le menu (menu décroché). Si *b* = 0, la première ligne du menu est une ligne insérée.  Les lignes (index) du menu sont numérotés à partir de zéro dans le cas *-tearoff 0*, et à partir de 1 dans le cas *-tearoff 1*.
Commandes les plus utilisées sur les menus	
add type *?options ?*	Ajoute une ligne en fin de menu, cela peut être une commande (*command*), un séparateur (*separator*), une cascade (*cascade*), un bouton radio (*radiobutton*) ou une boîte à cocher (*checkbutton*). Ces deux derniers types supportent les mêmes options que les *widget* de base du même nom. Si l'option *-accelerator un_raccourci* est spécifiée, alors lorsque ce raccourci est effectué, la ligne du menu associée est actionnée.
cget *option*	Renvoie la valeur de l'option spécifiée du menu
configure *?options ?*	Change les options spécifiées (voir plus haut) du menu
delete *index1 index2*	Efface les lignes de menu comprises entre les deux indexes fournis
entrycget *index*	Renvoie la valeur des options spécifiées sur la ligne dont l'index est donné
entryconfigure *index*	Configure une (des) option(s) de la ligne de menu spécifiée par

	index
insert *index type ?options ?*	Même chose que *add*, mais insère la nouvelle ligne à la position donnée

2. Les éléments de texte

Les widgets permettant d'afficher ou de saisir du texte et de manipuler des affichages textuels (polices, justification, coupure de mots, etc...) sont les suivants:

a) Les labels

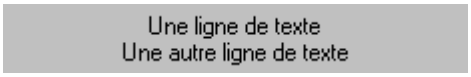
Un label 

(*label*) est un widget permettant d'afficher une ligne de texte non modifiable par l'utilisateur. Il peut afficher un texte statique ou bien le contenu d'une variable globale.
Exemples de créations de labels :

label *.top1.f.lab1* ***-text*** *"Ceci est un label"*
pack configure *.top1.f.lab1* ***-side top***
Création d'un label affichant "Ceci est un label"
.top1.f.lab1 ***configure -justify center***
Modification de la justification de son contenu

Options les plus utilisées pour les labels	
-image *img*	L'image *img* spécifiée est affichée sur le label
-justify *justification*	Le texte affiché dans le label est justifié de la façon spécifiée
-text *t*	*t* est le texte associé au label
-textvariable *v*	*v* est une variable globale dont le contenu est affiché sur le label, et dont toute modification entraîne la modification du texte du label
Commandes les plus utilisées sur les labels	
cget *option*	Renvoie la valeur de l'option spécifiée du label
configure *options*	Change les options spécifiées (voir plus haut) du label

b) Les messages

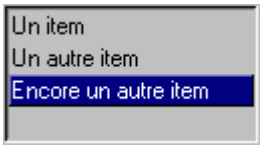
Un message 

(*message*) est un label (ne pouvant afficher d'image) permettant d'afficher un texte sur plusieurs lignes. Il coupera le texte automatiquement (en fin de mot quand c'est possible) si il dépasse la largeur de la fenêtre message ou on pourra insérer des *\n* pour le forcer à changer de ligne. La particularité d'un message est qu'il possède un *aspect* basé sur 100 lorsque l'on veut que le message soit aussi large que haut, 200 si on le veut deux fois plus large que haut, 50 quand on le veut deux fois plus haut que large etc... Par défaut celui-ci est à 150. Cet *aspect* complique l'utilisation de ce *widget*.
Exemples de création de messages:

```
message .top1.f.msg -text "Une ligne de texte\nUne autre ligne de texte" -aspect 300 \
   -justify center
pack configure .top1.f.msg -side left
# Création d'un message 3 fois plus large que haut.
message .top1.f.msg2 -text "Une ligne de texte\nUne autre ligne de texte" -aspect 300 \
   -justify center
pack configure .top1.f.msg2 -side right
# Ce message sera tout en hauteur (normalement 1 caractère par ligne)
```

Options les plus utilisées pour les messages	
-aspect *a*	Le texte est découpé entre caractères pour qu'il soit *a*/100 fois plus large que long.
-justify *justification*	Le texte affiché dans le message est justifié de la façon spécifiée
-text *t*	*t* est le texte associé au message
-textvariable *v*	*v* est une variable globale dont le contenu est affiché sur le message, et dont toute modification entraîne la modification du texte du message
Commandes les plus utilisées sur les messages	
cget *option*	Renvoie la valeur de l'option spécifiée du message
configure *options*	Change les options spécifiées (voir plus haut) du message

c) Les listes de texte



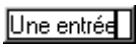
Une liste de texte (*listbox*) est une boîte contenant une liste de chaînes de caractères que l'utilisateur pourra sélectionner à l'aide du clavier ou de la souris. Le texte sélectionné est copié dans le presse-papier.
Exemples de création de listes de texte:

```
listbox .top1.f.lst -height 4
pack configure .top1.f.lst -side top
#Création d'une liste de texte qui peut contenir 4 éléments visibles à la fois
.top1.f.lst insert end "Un item"
.top1.f.lst insert end "Encore un autre item"
set t "Un autre item"
.top1.f.lst insert 1 $t
```

Options les plus utilisées pour les listes de texte	
-height *h*	Donne la longueur maximale d'éléments visibles à la fois dans la liste
-selectmode *mode*	Si *mode* = ***browse*** (par défaut) une seule ligne peut être sélectionnée. Si *mode* = ***multiple*** plusieurs lignes peuvent être sélectionnées: un click sur une ligne sélectionnée la déselectionne, un click sur une ligne non sélectionnée la sélectionne. Si *mode* = ***extended*** plusieurs lignes peuvent être sélectionnées: la sélection fonctionne à la manière des sélections sous windows: un click seul sélectionne 1 et 1 seule ligne, un click+shift sélectionne tous les

	éléments entre le dernier élément sélectionné et le click, ctrl+click sélectionne l'élément clické en plus des autres.
Commandes les plus utilisées sur les listes de texte	
activate *index*	L'élément n° *index* est sélectionné (0 correspond au premier élément)  Dans la suite, index est soit un entier, soit *end* pour le dernier élément+1, ou *active* pour l'index du dernier élément sélectionné
cget *option*	Retourne la valeur associée à l'option *option* spécifiée
configure *?options?*	Configure les options données
curselection	Renvoie la liste des indexes des éléments sélectionnés
delete *index1 index2*	Détruit les éléments compris entre les deux indexes donnés
get *index1 index2*	Renvoie la liste des valeurs contenues dans les lignes comprises entre les deux indexes
insert *index ?elts?*	Insère le ou les éléments sous forme de chaînes de caractères dans la liste à partir de la position sépcifiée par *index*

d) Les entrées de texte

Une entrée de texte (*entry*) 

est un label modifiable par l'utilisateur. Il ne peut (forcément) pas contenir d'image ni de texte statique => utilisation d'une variable globale associée par l'option *-textvariable*.
Exemples de création d'entrées de texte:
entry *.top1.f.e* ***-justify*** *left* ***-textvariable*** *E*
pack configure *.top1.f.e* ***-side top***
Création d'une entrée de texte associée à la variable E

Options les plus utilisées pour les entrées de texte	
-justify *justification*	Le texte affiché dans l'entrée est justifié de la façon spécifiée
-state *s*	L'entrée est disponible (*s* = ***normal***) ou indisponible (*s* = ***disabled***)
-textvariable *v*	*v* est une variable globale dont le contenu est affiché sur l'entrée, et dont toute modification entraîne la modification du texte de l'entrée. Bien sûr toute modification de l'entrée par l'utilisateur est immédiatement répercutée sur *v*
-width *w*	Largeur en nombre de caractères de l'entrée
Commandes les plus utilisées sur les entrées de texte	
cget *option*	Renvoie la valeur de l'option spécifiée de l'entrée
configure *?options?*	Change les options spécifiées (voir plus haut) de l'entrée

e) Le texte

Un texte (*text*) est un message modifiable par l'utilisateur. Il offre en plus davantage de fonctionnalités, et sa documentation est assez longue. Il peut, en plus du texte, contenir d'autres widgets. Il permet, par un système de *tags* associés à certains éléments créés (texte, boutons, ...) d'effectuer la même action sur tous les objets *taggés* par le même *tag* en une seule commande. *Tagger* un objet, c'est un peu comme l'assigner à un groupe, quand on effectue une action sur ce groupe, on l'effectue sur chaque objet le composant.

Un exemple, très simple, permettant l'affichage du contenu d'un fichier texte dans une fenêtre de texte:

```
text .top1.f.t
pack configure  .top1.f.t -side top -expand 1 -fill both
# Création d'un widget text
set F [open $FileName r]
#Ouverture du fichier en lecture
while {[eof $F]==0} {
# Jusqu'à la fin du fichier
 gets $F UneLigne
 # Lire une ligne du fichier
 .top1.f.t insert end $UneLigne
 .top1.f .t insert end "\n"
 # L'afficher dans le widget de texte
}
close $F
```

Options les plus utilisées pour les textes	
-height *h*	Le *widget* possèdera h lignes visibles.
-state *s*	Le texte est modifiable (*s* = ***normal***) ou non (*s* = ***disabled***)
-width *w*	Largeur visible en nombre de caractères du texte
-wrap *wr*	Le texte sera coupé si nécessaire en fin de ligne (*wr* = ***char***), en fin de ligne mais sur fin de mot (*wr* = ***word***), ou ne sera pas coupé (*wr* = ***none***)
Commandes les plus utilisées sur les textes	
cget *option*	Renvoie la valeur de l'option spécifiée du *text*
configure *?options?*	Change les options spécifiées du *text*
delete *index1 ?index2?*	Supprime le texte compris entre les deux indexes (voir la partie index pour la notation des indexes)
get *index1 ?index2?*	Renvoie le texte compris entre les deux indexes (voir la partie index pour la notation des indexes)
insert *index chars ?tags?*	Insère le texte spécifié à l'index spécifié taggé par les tags donnés (voir index)
Définition des indexes	
Les indexes sont donnés dans le format *base ?modificateurs?* où base peut être donnée sous la forme: *ligne.colonne*, ***@****x,y* où x et y, coordonnées en pixels, indiquent le caractère le plus proche, ***end***, la fin du texte, *nomtag.****first*** qui est le premier caractère taggé par *nomtag*, *nomtag.****last***, le dernier. Les modificateurs sont: *+ nb chars*, *- nb chars*, *+ nb lines*, *- nb lines*, où *nb* est un entier, ***linestart***, ***lineend***, ***wordstart***, ***wordend***.	

Il est à noter que les *widgets listbox*, *entry* et *text* offrent des fonctions de sélection/couper/copier/coller compatibles avec le système d'exploitation.

3. Un widget très puissant: le canvas

Le canvas est le *widget* offrant le plus de fonctionnalités. Il permet d'afficher des points, des lignes (pouvant être des flèches), des ovales, arcs, rectangles, polygones, images, du texte ainsi que n'importe quel autre widget (mis à part des fenêtre *toplevel*). Comme les éléments de texte, chaque élément créé peut être *taggé*, mais l'un des atouts principaux des *canvas* est que chaque élément, lors de sa création, se voit assigner un indice. En conservant l'indice des éléments intéressants, on pourra modifier très simplement l'allure des éléments dessinés qui deviennent ainsi des objets de canvas. Il est à noter que les coordonnées passées au *canvas* sont des réels.

Exemple d'utilisation de canvas:

canvas *.top1.f.c*
pack configure *.top.f.c* ***-side top -expand*** *1* ***-fill both***
#Création d'un canvas
lappend *ListeDitems [.top1.f.c* ***create rectangle12*** *15 45 32* ***-fill*** *red]*
#Création d'un rectangle rouge dans le canvas, son coin supérieur gauche est en 12,15 et son coin inférieur droit en 45,32. La commande *create* renvoie l'indice du rectangle dans le canvas. On peut par exemple conserver les objets créés dans une liste... Ici *ListeDitems*.
.top1.f.c ***itemconfigure*** *[****lindex*** *$ListeDitems [****expr*** *[****llength*** *$ListeDitems]-1]]* ***-fill*** *blue*
Le rectangle créé devient bleu

Options les plus utilisées pour les canvas	
-height *h*	Le *canvas* aura une hauteur de *h* pixels
-width *w*	Le *canvas* aura une largeur de *w* pixels
Commandes les plus utilisées sur les canvas	
cget *option*	Renvoie la valeur de l'option spécifiée du *canvas*
configure *?options?*	Change les options spécifiées du *canvas*
coords *TagOuId ?x0 y0?*	Si *x0* et *y0* ne sont pas spécifiés, renvoie les coordonnées de l'objet spécifié par son identificateur (renvoyé par *canvas create* lors de sa création) ou du premier objet *taggé* par le *tag*. S'ils sont spécifiés, modifie les coordonnées de l'objet spécifié
create *type x y ?x y? ?options?*	Crée en *x,y* un objet du type spécifié (voir types d'objets de canvas) avec les options spécifiées. Cet objet pourra cacher les objets créés avant lui si ils partagent des points: il se trouve devant eux. On pourra changer cela en utilisant les commandes *lower* et *raise*. Renvoie l'identificateur de l'objet créé
delete *TagOuId*	Supprime l'objet ou les objets désignés par le *tag* ou l'indentificateur *TagOuId*
itemcget *TagOuId option*	Retourne la valeur de l'*option* spécifiée pour l'objet dont l'identificateur est spécifié ou bien du premier objet *taggé* par le *tag* si *TagOuId* est un *tag*
itemconfigure *TagOuId ?options?*	Modifie la valeur des *options* spécifiées pour l'objet dont l'identificateur est spécifié ou bien du premier objet *taggé* par le *tag* si *TagOuId* est un *tag*

lower *TagOuId ?TagsOuIds?*	L'objet spécifié par l'identificateur *TagOuId* ou bien la liste des objets *taggés* par *TagOuId* si celui-ci est un *tag* est placé derrière les objets spécifiés par *TagsOuIds*
move *TagOuId dx dy*	Bouge l'objet (ou les objets si *TagOuId* est un *tag*) de *dx,dy* pixels
raise *TagOuId ?TagsOuIds?*	L'objet spécifié par l'identificateur *TagOuId* ou bien la liste des objets *taggés* par *TagOuId* si celui-ci est un *tag* est placé devant les objets spécifiés par *TagsOuIds*
scale *TagOuId x0 y0 xscale yscale*	Chaque point M=(x,y) dans le repère de centre (x0,y0) composant les objets spécifiés par *TagOuId* se retrouve en M'=(*xscale* x x, *yscale* x y). Dans le cas où (x0,y0)=(0,0), c'est un zoom de *xscale* en x, et de *yscale* en y.  Dans le cas des images et des *widgets* créés dans le *canvas*, *scale* ne modifie que les coordonnées de l'objet, pas son contenu: ainsi cette commande ne permettra pas de zoomer sur une image.
Types d'objet que l'on peut créer dans un canvas	
create arc *x1 y1 x2 y2 ?options?*	Un arc est défini d'abord par un rectangle pouvant le contenir, donné par deux de ses coins opposés: (*x1*,*y1*) et (*x2*,*y2*). La taille angulaire, par défaut 360 pour une ellipse, est donnée par les options ***-extent*** *degrés* où *degrés* donne la taille de la partie visible de l'ellipse et ***-start*** *degrés* où *degrés* donne le point de départ de l'arc. Nous aurons affaire à un fromage si on utilise l'option ***-fill*** *couleur*. Le trait du bord aura la couleur spécifiée par ***-outline*** *couleur* et l'épaisseur donnée par ***-width*** *largeur*. L'objet ainsi créé peut être *taggé* avec l'option ***-tag*** *ListeDeTags*.
create image *x y ?options?*	Place une image donnée par ***-image*** *NomImage* aux coordonnées (*x*,*y*). Il faut utiliser l'option ***-anchor*** *position* pour préciser si (*x*,*y*) doit être le milieu de l'image (*position* = ***center***) ou un des milieux des cotés de l'image (*position* = ***n***, ***s***, ***e***, ***w*** pour nord, sud, est, ouest) ou bien l'un des coins de l'image (*position* = ***nw*** , ***ne***, ***sw***, ***se*** pour nord-ouest, nord-est, sud-ouest, sud-est). L'objet ainsi créé peut être *taggé* avec l'option ***-tag*** *ListeDeTags*.
create line *x1 y1 ... xn yn ?options?*	Crée une ligne passant par les points de coordonnées (*xi*,*yi*). Son épaisseur est donnée par ***-width*** *épaisseur*, et sa couleur par ***-fill*** *couleur*. Elle peut être simple (*-arrow none*), posséder une flèche à la fin (***-arrow last***) ou au début (***-arrow first***) ou aux deux extrémités (***-arrow both***). Elle peut être dessinée en courbe de Bézier avec l'option *-smooth 1* avec un arrondi prenant en compte *nb* points pour ***-splinesteps*** *nb*. L'objet ainsi créé peut être *taggé* avec l'option ***-tag*** *ListeDeTags*.
create oval *x1 y1 x2 y2 ?options?*	Crée un ovale contenu dans le rectangle donné par les

	coordonnées (*x1*,*y1*) et (*x2*,*y2*) de deux de ses coins opposés. Sa couleur de remplissage est donnée par ***-fill*** *couleur* et la couleur de son bord est donnée par ***-outline*** *couleur*. L'épaisseur de son bord est donnée par ***-width*** *épaisseur*. L'objet ainsi créé peut être *taggé* avec l'option ***-tag*** *ListeDeTags*.
create polygon *x1 y1 ... xn yn ?options?*	Crée un polygone dont les sommets sont les points de coordonnées (*xi*,*yi*). Sa couleur est donnée par ***-fill*** *couleur*, l'épaisseur de son bord est donnée par ***-width*** *épaisseur*, et sa couleur par *-outline couleur*. Il peut être dessiné en courbe de Bézier avec l'option ***-smooth*** *1* avec un arrondi prenant en compte *nb* points pour ***-splinesteps*** *nb*. L'objet ainsi créé peut être *taggé* avec l'option ***-tag*** *ListeDeTags*.
create rectangle *x1 y1 x2 y2 ?options?*	Crée un rectangle ayant pour coins opposés (*x1*,*y1*) et (*x2*,*y2*) de couleur donnée par ***-fill*** *couleur*, et dont le bord a une épaisseur donnée par ***-width*** *épaisseur* et une couleur donnée par ***-outline*** *couleur*. L'objet ainsi créé peut être *taggé* avec l'option ***-tag*** *ListeDeTags*.
create text *x y ?options?*	Crée du texte dont la position relative à (*x*,*y*) est donnée par ***-anchor*** *position*: pour préciser si (*x*,*y*) doit être le milieu du texte (*position* = ***center***) ou un des milieux des cotés du texte (*position* = ***n***, ***s***, ***e***, ***w*** pour nord, sud, est, ouest) ou bien l'un des coins du texte (*position* = ***nw*** , ***ne***, ***sw***, ***se*** pour nord-ouest, nord-est, sud-ouest, sud-est). La couleur des caractères est donnée par ***-fill*** *couleur*, leur police par ***-font*** *police*. Le texte peut être justifié par ***-justify*** *justification* et être coupé au bout de *nb* caractères par ***-width*** *nb*. Le texte affiché est donné par ***-text*** *texte*. L'objet ainsi créé peut être *taggé* avec l'option ***-tag*** *ListeDeTags*.
create window *x y ?options?*	Place un *widget* existant donné par ***-window*** *widget* dans le canvas. Sa position relative à (*x*,*y*) est donnée par ***-anchor*** *position*: pour préciser si (*x*,*y*) doit être le milieu du *widget* (*position* = ***center***) ou un des milieux des cotés du *widget* (*position* = ***n***, ***s***, ***e***, ***w*** pour nord, sud, est, ouest) ou bien l'un des coins du *widget* (*position* = ***nw*** , ***ne***, ***sw***, ***se*** pour nord-ouest, nord-est, sud-ouest, sud-est). On peut spécifier sa hauteur en pixels ainsi que sa largeur par ***-height*** *pixels* et ***-width*** *pixels*. L'objet ainsi créé peut être *taggé* avec l'option ***-tag*** *ListeDeTags*.  Il est impossible de placer une *toplevel* dans un canvas, et le *widget* placé doit être soit un fils soit le fils d'un des ancêtres du canvas.

4. Les barres de défilement et règles

Tk propose encore deux types de *widgets*:

a) Les règles

Les règles (*scale*) 

, comme chaque *widget* peuvent avoir une foule d'options. Le principe de son utilisation est d'associer la valeur de la règle à une variable et d'agir en conséquence. Il est possible de donner une borne inférieure et supérieure à la valeur que peut prendre la règle, ainsi qu'une précision. Ainsi chaque déplacement de la règle la positionnera sur un multiple du nombre défini comme la précision.

Voici un exemple de création de règle:
scale *.top1.f.sc* ***-orient horizontal -variable*** *V* ***-from*** *0* ***-to*** *100* ***-resolution*** *1*
pack configure *.top1.f.sc* ***-side top -fill x -expand 1***
Création d'une règle associée à la variable V. Tout changement de V influe sur la règle
et tout changement de la règle influe sur V

Options les plus utilisées pour les règles	
-command *com*	A chaque modification de la règle la commande *com* est appelée. Tcl/Tk ajoute à la fin de *com* la valeur correspondant à la position de la règle.
-from *valeur*	La règle ne peut pas prendre une valeur en dessous de *valeur*.
-label *text*	Le texte *text* est affiché sur (resp. à côté) la règle si elle est verticale (resp. horizontale).
-length *pixels*	La règle a une longueur de *pixels* pixels.
-orient *orientation*	La règle est verticale (**vertical**) ou horizontale (**horizontal**)
-resolution *précision*	La règle prend des valeurs multiples de *précision*.
-showvalue *bool*	Si *bool* est à 1, la valeur de la règle est affichée.
-state *s*	La règle est modifiable (*s* = ***normal***) ou non (*s* = ***disabled***)
-to *valeur*	La règle ne peut pas prendre une valeur au dessus de *valeur*.
-variable *v*	*v* est une variable globale dont le contenu correspond à la position de la règle, et dont toute modification entraîne la modification de la règle. Bien sûr toute modification de la règle par l'utilisateur est immédiatement répercutée sur *v*
-width *w*	La largeur de la règle est *w*
Commandes les plus utilisées sur les règles	
cget *option*	Renvoie la valeur de l'option spécifiée
configure *?options?*	Change les options spécifiées (voir plus haut) de la règle

b) Les barres de défilement

Les barres de défilement (*scrollbar*)  , horizontales ou verticales, permettent de faire défiler les *listbox*, les *canvas* et *text* dans les deux directions, et les *entry* dans le sens horizontal. Ce sont des *widgets* que l'on associe en général à un des *widgets* ci-

dessus par l'option *-command nom_widget_a_scroller set*. Il est important de comprendre que le comportement d'une barre de défilement est non seulement défini par elle-même (avec les options spécifiées sur elle) mais aussi par le *widget* qu'elle permet de scroller. En effet pour les *widgets* qui seront *scrollés*, certaines options vont être employées pour borner les déplacements de la *scrollbar* ou bien pour bouger celle-ci lorsque la vue du *widget* est changée.

Voici un exemple de canvas *scrollable* dans les deux sens. Il est important de noter que pour permettre à une *scrollbar* d'être à droite du *canvas*, et à l'autre d'être en dessous des deux, nous créons une frame qui va contenir le *canvas* et une *scrollbar* à aligner (voir III.A.2).

```
frame .top1.f.f
pack configure .top1.f.f -side left -fill both -expand 1
# frame hiérarchisante qui permet d'aligner le canvas et la scrollbar verticale
scrollbar .top1.f.f.scrollv -orient vertical -command ".top1.f.f.canvas yview"
pack configure .top1.f.f.scrollv -side right -fill y -expand 1
# Création de la scrollbar verticale, commandant la vue verticale (yview) du canvas
# Remarque: yview prend 2 arguments, qui sont passés automatiquement. Car yview, lorsque
# la scrollbar sera bougée, aura pour argument deux flottants compris entre 0 et 1.
# 0 représente le haut du widget à scroller, 1 le bas, et par exemple 0.333 le tiers de la hauteur
# du widget. Les deux arguments passés à yview automatiquement sont le haut de la partie visible
# du widget, et le bas de la partie visible.
scrollbar .top1.f.scrollh -orient horizontal -command ".top1.f.f.canvas xview"
pack configure .top1.f.scrollh -side bottom -fill x -expand 1
# Création de la scrollbar horizontale. Note: elle n'est pas créée dans la sous-frame.
canvas .top1.f.f.canvas -scrollregion "0 0 1000 600" -yscrollcommand ".top1.f.f.scrollv set" \
-xscrollcommand ".top1.f.scrollh set"
pack configure .top1.f.f -side top -expand 1 -fill both
# Le canvas est créé, il aura une région visible définie par la scrollregion, lorsque sa vue interne
# sera changée, la scrollbar bougera grâce aux deux options -x(ou y)scrollcommand qui mettent
# à jour l'apparence des scrollbars. Comme yview plus haut, ici set se voit automatiquement
# adjoindre des arguments qui dépendent de l'action effectuée sur la scrollbar. Nous ne les
# détaillons pas ici car ils sont transparents pour le programmeur.
```

Et voilà en 7 lignes de code, un canvas *scrollable* dans les deux sens.

Options les plus utilisées pour les barres de défilement	
-command *com*	A chaque fois que l'utilisateur fait défiler la *scrollbar*, la commande *com* est appelée avec des arguments passés automatiquement. En général *Com* est *Nom_du_widget_à_scroller xview_ou_yview*.
-orient *sens*	La barre de défilement est horizontale (*sens* = **horizontal**) ou verticale (*sens* = **vertical**).
Commandes les plus utilisées sur les barres de défilement	
cget *option*	Renvoie la valeur de l'option spécifiée
configure *?options?*	Change les options spécifiées (voir plus haut) de la barre
Options à utiliser sur les *widgets* défilants	

options de **canvas**	Les options ***-xscrollcommand*** *Nom_scrollbar* ***set*** et ***-yscrollcommand*** *Nom_scrollbar* ***set*** doivent être précisées pour que la *scrollbar* soit mise à jour lorsque la vue du *canvas* change (utilisation de la commande ***xview*** ou ***yview*** sur le canvas). De plus on peut spécifier une zone pouvant être scrollée par ***-scrollregion*** *"x0 y0 x1 y1"*où les coordonnées définissent un rectangle devant être visible via *scrolling*.
options de **listbox**	Les options ***-xscrollcommand*** et ***-yscrollcommand*** sont utilisées de la même façon que pour les *canvas* mais la *scrollregion* est calculée automatiquement grâce aux options ***-width*** et ***-height***..
options de **text**	Les options ***-xscrollcommand*** et ***-yscrollcommand*** sont utilisées de la même façon que pour les *canvas* mais la *scrollregion* est calculée automatiquement grâce aux options ***-width*** et ***-height***.
options de **entry**	L'option ***-xscrollcommand*** est utilisée (les *entry* ne supportent que le défilement horizontale) et la *scrollregion* est définie par l'option ***-width*** *nb_caractères*.

C. Généralités sur les widgets

Vous avez sans doute remarqué que les widgets pouvaient être modifiés après leur création, par exemple la plupart du temps par la commande *configure*, mais aussi souvent par la commande *add* comme pour les menus.

En fait, lorsqu'un widget .top.monwidget est créé, une procédure .top.monwidget est créée, qui peut être appelée avec différents paramètres (*configure*, *add*,...). Il peut être utile, lorsque tous les événements ne peuvent pas êtres récupérés sur ce widget de « biaiser » la procédure créée. Prenons un exemple : nous avons un widget de texte et nous aimerions savoir quand du texte y est inséré ou supprimé. Il n'y a à priori aucun moyen de le savoir sauf en récupérant chaque caractère tapé et chaque clique souris. De là un seul moyen : essayer de deviner comment le widget se comportera face à tel ou tel événement. Le problème est qu'il n'y a rien de moins portable, que ce soit entre les différences de plate-forme ou de version de Tcl/Tk. La solution qui permet d'y remédier utilise la commande ***rename*** sur la procédure créée pour le widget. Par exemple :

```
text .t
pack configure .t
rename .t mon.t
# la procédure appelée .t est renommée mon.t
proc .t {args} {
  switch [lindex $args 0] {
    insert {
     # insertion de caractères
    }
    delete {
      # suppression de caractères
    }
  }
  eval "mon.t $args"
  # Appel de la procédure originelle du widget .t
}
```

IV. Programmation avancée

Voilà cette liste exhaustive des *widgets* de base de Tk terminée. C'est bien mais, bon, ça ne dit pas comment programmer un menu déroulant, un menu contextuel, une barre d'outils, une barre d'état, et un canvas dans lequel on peut insérer des objets via un menu contextuel, etc...
C'est ce que nous allons expliquer dans ce chapitre.

A. Modularité

Comme pour tout langage, il est bon de modulariser un programme Tcl/Tk le plus possible. La méthode la plus simple pour y arriver est de découper son programme non seulement en fonctions mais aussi en fichiers, car on s'aperçoit vite que pas mal de modules s'avèrent utiles à plusieurs programmes (bibliothèques de *widgets* de haut niveau, ...).
La méthodologie conseillée est la suivante:

Créer un fichier source qui sera chargé de faire interpréter les différents fichiers Tcl/Tk utiles à l'application pour fonctionner. Pour cela utiliser la commande ***source*** *nom_fichier*.

Les séparateurs pour les noms de répertoire étant différents suivant les systèmes d'exploitation, utiliser les commandes ***file join***, ***file root***, etc... (voir l'aide sur ***file***) pour manipuler les noms de fichier.

Il faut penser que les fichiers Tcl/Tk ne seront pas forcément appelés du répertoire où ils se trouvent. Il est donc nécessaire, pour trouver les fichiers utiles à l'application (fichiers source, images...) de conserver quelque part le chemin absolu du fichier principal en exécutant dès le lancement de l'application:

set *CheminAbsolu* ***[info script]***
CheminAbsolu, une variable globale contient le nom (au pire relatif, par exemple
/mesprogs/toto.tcl sous MsWindows) du fichier principal
set *CheminAbsolu* ***[file dirname*** *$CheminAbsolu****]***
CheminAbsolu contient maintenant le chemin du fichier principal
source [file join *$CheminAbsolu lib UnFichierSource.tcl****]***
Interprète un fichier source Tcl/Tk situé dans le répertoire lib sous le fichier principal

Ensuite on chargera les images utilisées par l'application, par exemple pour la barre d'outils. Si celles-ci sont dans le répertoire "images" sous le fichier principal, par exemple:
image create photo *ImageOuvrirFichier* ***-file [file join $****CheminAbsolu images ouvrir.gif****]*** \
-format gif -height *32* ***-width*** *32*
L'image *ImageOuvrirFichier* contient maintenant l'image gif contenue dans *ouvrir.gif*

Il reste à créer les fonctions utilisées et notre (nos) fenêtre(s). En général, nous appellerons par raison de commodité la fonction créant la ou les fenêtres *ShowWindow*. Voici à quoi elle ressemblera dans le cas général:

```
proc ShowWindow {} {
  wm positionfrom  . user
  wm sizefrom . user
  wm title . {Titre de l'application}
  wm protocol . WM_DELETE_WINDOW Quit
 # Avec Quit une fonction définie dans l'application
 frame .f
 pack configure .f -fill both -expand 1
 # Création de la frame de plus haut niveau
 pack configure [ShowMenu .f left right] -side top -fill x -expand 1
 pack configure [ShowStateBar .f] -side bottom -fill x -expand 1
 pack configure [ShowToolBar .f left left] -side top -fill x -expand 1
 pack configure [ShowCanvas .f] -side top -fill both -expand 1
 # Appel des fonctions créant le menu déroulant, le canvas, la barre d'état et la barre d'outils
 # Ces fonctions sont définies dans les paragraphes suivants et renvoie leur nom
 # pour permettre à ShowWindow de les packer où il faut. Remarquez le passage du
# widget .f qui sera le père des widgets créés par ces fonctions.
 PrepareContextualMenu
 # Fonction qui créera un menu contextuel
}
```

Les fonctions appelées peuvent être définies dans d'autres fichiers.

B. Widgets de haut niveau

1. Menu déroulant

Il est intéressant de pouvoir facilement créer un menu en haut ou en bas de la fenêtre, à l'horizontale, ou bien à droite ou à gauche de la fenêtre, à la verticale. C'est pour cela que l'on va différer le *packing* du menu à l'extérieur de la fonction, et que l'on va paramétrer le positionnement des boutons de menu les uns par rapport aux autres.

```
proc ShowMenu {w side_to_pack side_to_pack_help} {
 # Crée un menu déroulant dans le widget w passé en argument
 # Les boutons seront packés du côté indiqué par side_to_pack, sauf le bouton "Help" qui sera
 # packé du côté indiqué par side_to_pack_help
 frame $w.menu
 # Création de la frame dans laquelle nous mettrons le menu déroulant
 # Son packing aura lieu dans la fonction appelante
 ############### CREATION DU MENU File ###############
 menubutton $w.menu.file -menu "$w.menu.file.m" -text {File} -underline {0}
 pack configure $w.menu.file -side $side_to_pack
 # Création d'un bouton de menu associé au menu $w.menu.file.m. packé comme spécifié
 # par l'argument. Normalement c'est un alignement gauche mais, sait-on jamais
 menu $w.menu.file.m
 w.menu.file.m add command -command {New} -underline {0} -label {New}
 # Menu New associé à l'appel de la fonction New à définir
```

```
$w.menu.file.m add command -command {Open} -underline {0} -label {Open...}
# Menu Open associé à l'appel de la fonction Open à définir
$w.menu.file.m add separator
$w.menu.file.m add command -command {Save} -underline {0} -label {Save}
# Menu Save associé à l'appel de la fonction Save à définir
$w.menu.file.m add command -command {SaveAs} -underline {5} -label {Save as...}
# Menu Save as associé à l'appel de la fonction SaveAs à définir
$w.menu.file.m add separator
$w.menu.file.m add command -command {Quit} -underline {1} -label {Exit}
# Menu Exit associé à l'appel de la fonction Quit à définir
############ CREATION DU  MENU Help ##############
menubutton $w.menu.help -menu "$w.menu.help.m" -text {Help} -underline {0}
pack configure $w.menu.help -side $side_to_pack_help
# Création du menu Help
menu $w.menu.help
$w.menu.help.m add command -command {About} -underline {0} -label {About...}
# Menu About associé à l'appel de la fonction About à définir
$w.menu.help.m add command -command {Help} -underline {0} -label {Help}
# Menu Help associé à l'appel de la fonction Help à définir
return $w.menu
#Renvoie le nom de sa frame de plus haut niveau pour permettre à la fonction appelante
# de packer le menu. D'habitude il est en haut mais on peut l'imaginer ailleurs.
}
```

2. Barre d'état (ou d'aide en ligne)

Il est intéressant de mettre en bas d'une application une barre d'aide en ligne qui contiendra normalement le nom du fichier ouvert, mais qui lorsque la souris passera sur un bouton, ou sur un objet de l'application, renseignera l'utilisateur sur ce que le bouton permet de faire ou bien sur l'objet. La récupération d'événements changeant l'état de la barre d'état (souris passe sur un objet...) sera expliquée dans la section ***bindings***.

Le contenu de cette barre sera dans une pile. Ainsi lorsque la souris est au-dessus d'un bouton, un texte explicatif est affiché, et lorsqu'elle sort, l'ancien texte, conservé dans la pile, est restauré.

La barre d'état sera un label associé à une variable globale, et 2 fonctions (*PushState* et *PopState*) permettront de gérer la pile associée. Typiquement lorsque la souris arrive au dessus d'un objet, *PushState* est appelée, et lorsque la souris quitte la zone de l'objet, *PopState* est appelée.

```
proc ShowStateBar {w} {
# Montre une barre d'état fille de w
global StateText
# Variable globale associée à la barre d'état
label $w.state -textvariable StateText -borderwidth 2 -relief sunken
# Création du label qui contiendra l'aide en ligne
return $w.state
}
```

Voici les fonctions de gestion de la pile:

```
proc PushState {msg} {
 # msg est un texte à afficher dans la barre d'état
 global StateText StatePile TopStatePile
 # Les variables globales sont: la variable liée à la barre d'état, la pile de mots de la barre,
 # et l'indice du sommet de la pile: en effet celle-ci sera implémentée sous forme de tableau
 # pour plus de rapidité
 if {[info exists TopStatePile]==0} {
 # c'est la première utilisation de la pile
  set TopStatePile 0
 }
 incr TopStatePile 1
 set StatePile($TopStatePile) $msg
 # Mémorisation du message dans la pile
 set StateText $msg
 # Affichage du message dans la barre d'état
}
proc PopState {} {
 # Le message affiché dans la barre d'état est à enlever
 global StateText StatePile TopStatePile
if {[info exists TopStatePile] != 0} {
 # La pile existe
  if {$TopStatePile > 1} {
  # La pile ne sera pas vide après suppression de l'élément
   unset StatePile($TopStatePile)
   incr TopStatePile -1
   set StateText $StatePile($TopStatePile)
  }
 }
}
```

3. Le *binding*: lier un évènement à une commande

Comme il est dit dans le paragraphe précédent décrivant la barre d'état, il peut être intéressant de lier certains événements à une commande (ainsi il faudra lier le passage de la souris sur un bouton à la fonction PushState, et sa sortie à la fonction PopState). Pour cela Tcl/Tk propose la commande:

bind *widget_ou_classe_de_widgets_ou_all Sequence_d_actions commande*

Si événement *Sequence_d_actions* a lieu sur *widget_ou_classe_de_widgets_ou_all*, alors *commande* est exécutée. *widget_ou_classe_de_widgets_ou_all* contient le nom d'un *widget* (par exemple *.top1.f.bouton1*), le nom d'une classe de *widgets* (par exemple *Button*) ou le mot clé *all* qui désigne tous les *widgets*. Remarque: si cet élément est une fenêtre *toplevel*, le *binding* s'applique à tous les fils de cette fenêtre.

Si une autre *commande* était déjà liée à l'événement *Sequence_d_actions*, alors elle est remplacée par cette *commande*, sauf si *commande* est précédée du symbole "+".

Séquence_d_actions est une action ou une suite d'actions. Lorsque c'est une suite d'action, l'événement arrive lorsque chaque action a eu lieu, dans l'ordre de la *Séquence_d_actions*.
Il est possible aussi d'appliquer un *binding* à des objets d'un canvas par la commande:
nom_canvas ***bind*** *tag_ou_identificateur_d_objet Séquence_d_actions Commande*
Les formes possibles pour une action sont <modificateur-modificateur-type-détail> où les types, les modificateurs et les détails sont définis dans le tableau suivant:

Types d'événements les plus fréquemment utilisés	
ButtonPress (ou ***Button***)	Arrive lorsqu'un bouton de la souris est pressé, si *détail* est spécifié (n° de bouton), alors l'événement arrive quand ce bouton est pressé.  Sous ms-windows, le bouton droit est le bouton 3. Sous macintosh il n'y a qu'un seul bouton.
ButtonRelease	Même chose que ***ButtonPress*** sauf que là événement arrive lorsque le bouton est lâché.
KeyPress	Arrive lorsqu'une touche, éventuellement spécifiée derrière (*KeyPress-Return* pour la touche retour chariot), est pressée.
KeyRelease	Même chose que ***KeyPress*** sauf qu'ici, l'événement arrive lorsque la touche est lâchée.
FocusIn	L'événement arrive lorsque le (ou les) *widget* ou l'objet spécifié est sélectionné.
FocusOut	L'événement arrive lorsque le (ou les) *widget* ou l'objet spécifié est désélectionné.
Enter	L'événement arrive lorsque la souris "entre" au dessus du (ou des) *widget* ou de l'objet spécifié.
Leave	L'événement arrive lorsque la souris "sort" du dessus du (ou des) *widget* ou de l'objet spécifié.
Motion	L'événement arrive lorsque la souris bouge dans le (les) *widget* ou l'objet spécifié. Le plus souvent, cet événement est accompagné d'un *modificateur*. Par exemple *B1-Motion* a lieu quand le bouton gauche est pressé et que la souris bouge.
Modificateurs les plus utilisés	
Control, ***Shift***, ***Alt***	La touche donnée doit être pressée
Button1, ***Button2***, ***Button3***, ***Button4***, ***Button5***, ou ***B1***, ***B2***, ***B3***, ***B4***, ***B5***	Le bouton spécifié doit être pressé.
Double, ***Triple***	L'événement doit avoir lieu deux (respectivement 3) fois. **<*Double-Button-1*>** est équivalent à **<*Button-1*><*Button-1*>**.

Des champs spéciaux peuvent être utilisés dans *Commande*. Ils sont définis dans le tableau suivant:

Champs pouvant être utilisés dans une commande de *binding*	
%k	Contient le code ascii de la touche pressée ou lâchée lors d'un *binding* ***KeyPress*** ou ***KeyRelease***.
%K	Contient le nom de la touche pressée ou lâchée lors d'un *binding* ***KeyPress*** ou ***KeyRelease***. Par exemple, pour la touche "a", contient "a", pour la

	touche retour chariot contient *Return*...
%A	Contient le caractère affichable par la touche pressée ou lâchée lors d'un *binding* ***KeyPress*** ou ***KeyRelease***. Par exemple, pour la touche "a", contient "a", pour la touche retour chariot ne contient rien....
%x	Contient la position de la souris en abscisse lorsque l'événement (qui doit être en rapport avec la souris) est arrivé. Cette position est donnée relativement au *widget* concerné.
%y	Contient la position de la souris en ordonnée lorsque l'événement (qui doit être en rapport avec la souris) est arrivé. Cette position est donnée relativement au *widget* concerné.
%X	Contient la position de la souris en abscisse lorsque l'événement (qui doit être en rapport avec la souris) est arrivé. Cette position est donnée relativement à l'écran.
%Y	Contient la position de la souris en ordonnée lorsque l'événement (qui doit être en rapport avec la souris) est arrivé. Cette position est donnée relativement à *l'écran.*
%W	Contient le nom du *widget* sur lequel l'événement est arrivé.

4. Une barre d'outils détachable

Ce paragraphe montre comment construire une barre d'outils à partir de petites images, de préférence au format gif. Leur taille sera de préférence comprise entre 20x20 et 32x32.

Le principe est de construire une frame dans laquelle il y a plusieurs sous-frames, correspondant chacune à une catégorie d'outils. Par exemple, la première sous-frame contient des outils liés à des commandes du menu Fichier (Nouveau, Ouvrir, Enregistrer), la seconde des outils de visualisation (Zoom...), ..., la dernière des outils d'aide.

```
proc ShowToolBar {w packing souspacking} {
 # Crée une barre d'outils avec bulles d'aide et aide contextuelle dans le widget w
 # packing est le packing des sous-frames les unes par rapport aux autres, et souspacking
 # est le packing des boutons des sous-frames les uns par rapport aux autres
 # On suppose que toutes les images ont été créées au début
 frame $w.tools
 frame $w.tools.file
 pack configure $w.tools.file -side $packing -ipadx 5
 button $w.tools.file.open -command {Open} -image {ImageOuvrirFichier}
 pack configure $w.tools.file.open -side $souspacking
 menu $w.tools.file.open.m -tearoff 0 -foreground black -background yellow \
  -borderwidth 1 -activeforeground black -activebackground yellow -activeborderwidth 0
# .....................
 # etc... pour tous les boutons et les sous-frames
 return $w.tools
}
```

Remarque: grâce aux paramétrages du packing, on peut imaginer un bouton dans la barre d'outils qui va décrocher ou raccrocher la barre d'outils à la fenêtre principale. Par

exemple pour décrocher la barre d'outils et la mettre dans une *toplevel .top1* avec les sous-frames de boutons les unes au dessous des autres:
destroy $w.tools
ShowToolBar .top1 top left

Voilà, pour avoir tous les éléments d'une application réussie, il ne nous reste plus qu'à créer un *canvas* dans lequel nous allons créer par exemple des rectangles, les déplacer, et les colorer à l'aide de menus contextuels.

5. Canvas *scrollable*

Construisons notre *canvas* à l'intérieur de la fonction *ShowCanvas*. La construction est identique à celle présentée en *III.B.4.b*.

```
proc ShowCanvas {w} {
 #Construit un canvas scrollable à l'intérieur du widget w
 frame $w.canvas
 scrollbar $w.canvas.scrollv -orient vertical -command "$w.canvas.f.c yview"
 pack configure $w.canvas.scrollv -side right -fill y -expand 1
 # Scrollbar verticale
 frame $w.canvas.f
 pack configure $w.canvas.f -side left -fill both -expand 1
 # Sous-frame pour packer la scrollbar horizontale et le canvas
 scrollbar $w.canvas.f.scrollh -orient vertical -command "$w.canvas.f.c xview"
 pack configure $w.canvas.f.scrollh -side bottom -fill x -expand 1
 # Création de la scrollbar horizontale
 canvas $w.canvas.f.c -scrollregion "0 0 1024 768" \
  -yscrollcommand "$w.canvas.scrollv set" -xscrollcommand "$w.canvas.f.scrollh set"
 pack configure $w.canvas.f.c -side top -fill both -expand 1
 global Canvas
 set Canvas $w.canvas.f.c
 #Création du canvas. En général, comme pas mal de fonctions auront à utiliser le canvas,
 # on met son nom dans une variable globale, ici Canvas.
 global tcl_platform
 if {[string compare $tcl_platform(platform) mac]==0} {
 # On détermine quel binding appliquer au menu contextuel suivant la plateforme
  set binding <Shift-ButtonPress>
 } else {
 set binding <ButtonPress-3>
 }
 bind $Canvas $binding "CanvasContextualMenu %X %Y %x %y"
 # Le menu contextuel sera mis à jour puis affiché par la fonction "CanvasContextualMenu"
 # à écrire...
 return $w.canvas
}
```

6. Menu contextuel

Il est intéressant d'ajouter les commandes usuelles du menu déroulant sur le canvas dans un menu contextuel: on clique sur le bouton droit (ou sur le bouton et pomme dans le cas du macintosh) et un menu contextuel apparaît, différent suivant l'endroit où l'on se trouve. Par exemple le menu contextuel du canvas va avoir un menu permettant de créer un rectangle. Le menu contextuel d'un rectangle va permettre de le colorer (extérieur et intérieur) ou de le supprimer. Le principe est de créer un type de menu contextuel pour le canvas, et un type de menu contextuel pour les rectangle. Les commandes associées aux menus sont différentes, pour le canvas, suivant l'endroit du click, qui va donner la position à laquelle créer le rectangle. Et suivant le rectangle, puisque c'est sur lui que vont avoir lieu les modifications, les commandes associées au menu contextuel du rectangle vont changer. Nous utiliserons donc :

```
proc PrepareContextualMenu {} {
 # Crée les menus contextuels qui pourront ensuite être modifiés
 menu .canvasmenu -tearoff 0
 .canvasmenu add command -label {Create rectangle}
 # Crée le menu contextuel appelé sur le canvas, l'entrée "Create rectangle" ne sera associée
 # à une commande qu'à chaque fois que le menu sera déroulé par la fonction à écrire
 # CanvasContextualMenu
 menu .rectanglemenu -tearoff 0
 .rectanglemenu add command -label {Change border color}
 .rectanglemenu add command -label {Change fill color}
 .rectanglemenu add command -label {Delete}
 # Crée le menu contextuel appelé sur les rectangles, les entrées seront associées
 # à des commandes à chaque fois que le menu sera déroulé par la fonction à écrire
 # RectangleContextualMenu
}
```

Ecrivons maintenant la fonction permettant d'afficher un menu contextuel dans le canvas:

```
proc CanvasContextualMenu {X Y x y} {
 # Affiche le menu contextuel du canvas en X,Y relativement à l'écran. Attention, comme le
 # canvas est scrollbable, il faut transformer x et y pour avoir les coordonnées canvas
 # associées.
 global Canvas
 # Le nom du canvas
 set Canvasx [$Canvas canvasx $x]
 set Canvasy [$Canvas canvasy $y]
 # Transforme x,y coordonnées relatives au widget canvas en coordonnées relatives au
 # canvas scrollable
 .canvasmenu entryconfigure 0 -command "CreateRectangle $Canvasx $Canvasy"
 # La première entrée du canvas est associée à la création d'un rectangle
 .canvasmenu post $X $Y
 # Affiche le menu contextuel
}
```

Voyons maintenant comment créer un rectangle qui sera lié à un menu contextuel:

```
proc CreateRectangle {x y} {
 # Crée un rectangle10x10 centré en x,y
 global Canvas
 set id [$Canvas create rectangle [expr $x -5] [expr $y -5] [expr $x+5] [expr $y+5]]
 # id contient l'identificateur du rectngle créé
 global tcl_platform
 if {[string compare $tcl_platform(platform) mac]==0} {
 # On détermine quel binding appliquer au menu contextuel suivant la plateforme
  set binding <Shift-ButtonPress>
 } else {
 set binding <ButtonPress-3>
 }
 $Canvas bind $id $binding "RectangleContextualMenu $id %X %Y"
 # Le menu contextuel sera configuré et affiché par la fonction à écrire
 # "RectangleContextualMenu"
}
```

Il ne reste plus qu'à écrire la fonction permettant d'afficher le menu contextuel associé aux rectangles:

```
proc RectangleContextualMenu {IdRectangle X Y} {
 # Déroule en X,Y un menu contextuel associé au rectangle dont l'identificateur est
 # IdRectangle
 global Canvas
 .rectanglemenu entryconfigure 0 -command "ChangeColor $IdRectangle {-color}"
 # ChangeColor est une fonction à écrire se basant sur le paragraphe "Changer une couleur"
 .rectanglemenu entryconfigure 1 -command "ChangeColor $IdRectangle {-fill}"
 .rectanglemenu entryconfigure 2 -command "$Canvas delete $IdRectangle"
 .rectanglemenu post $X $Y
}
```

C. Des commandes Tk bien utiles

Maintenant que la fenêtre principale est créée, il reste à utiliser des boîtes de dialogues pour réagir à certains événements. Tk en propose quelques unes, très pratiques.

1. Ouvrir, enregistrer un fichier

Tk propose des commandes ouvrant automatiquement une boîte de dialogue standard sur le système (à partir de la version Tcl7.6/Tk4.2). Celle-ci permet de sélectionner un fichier à ouvrir ou de sélectionner un nom de fichier pour un fichier à enregistrer et a le *look* de la plate-forme sur laquelle l'application tourne. Ces commandes sont:
tk_getOpenFile et **tk_getSaveFile** qui renvoient un nom de fichier, qui peut être vide si l'utilisateur n'a rien sélectionné.

2. Changer une couleur

De la même façon que les commandes ouvrant une boîte de dialogue standard pour sélectionner un nom de fichier, la commande **tk_chooseColor** permet de choisir une couleur. La chaîne renvoyée par cette fonction contient soit une couleur, soit une chaîne vide si l'utilisateur n'a pas choisi de couleur.

3. Boîtes de dialogue standard

Tk propose deux commandes permettant de définir simplement des boîtes de dialogue standard qui contiennent un message et un certain nombre de boutons, ainsi qu'une icône informant l'utilisateur sur le contenu du message (danger, information, question, erreur).
Ces commandes sont **tk_dialog** et **tk_messageBox**.

4. Boîte de dialogue personnalisée

Une boîte de dialogue doit souvent attendre une réponse, en interdisant tout accès à un autre objet de l'application, avant de rendre la main à celle-ci. En général la boîte de dialogue sera construite dans une fenêtre *toplevel*, et celle-ci ne rendra la main à l'application que lorsqu'elle sera détruite (par une réponse de l'utilisateur). Il faut dire explicitement à Tk de ne rien faire d'autre tant qu'elle n'est pas fermée par les commandes

grab set *nom_toplevel_contenant_boîte_de_dialogue*
qui demande à l'application de rester sur la fenêtre, et
tkwait window *nom_toplevel_contenant_boîte_de_dialogue*
qui demande à l'application d'attendre la destruction de la fenêtre.

Pour exemple, construisons une boîte de dialogue qui demandera à un utilisateur son nom et son mot de passe et renverra soit une chaîne vide si l'utilisateur a annulé, soit une liste contenant le nom et le mot de passe.

```
proc GetPassword {} {
 global getPassword
 # getPassword est un tableau qui va contenir 3 entrées:
 # name qui va contenir le nom de l'utilisateur
 # passwd qui va contenir son mot de passe
 # result qui va contenir 1 si et seulement si l'utilisateur a cliqué sur Ok
 set getPassword(result) 0
 set getPassword(name) ""
 set getPassword(passwd) ""
 toplevel .passwd
 wm title .passwd "Password"
 wm maxsize .passwd 200 100
 wm minsize .passwd 200 100
 wm positionfrom .passwd user
 wm sizefrom .passwd user
 frame .passwd.f
```

```
pack configure .passwd.f -side top -fill both -expand 1
frame .passwd.f.name
pack configure .passwd.f.name -side top -fill x
# Frame qui va contenir le label "Enter your name:" et une entrée pour le rentrer
label .passwd.f.name.l -text {Enter your name:}
pack configure .passwd.f.name.l -side left
entry .passwd.f.name.e -textvariable getPassword(name)
pack configure .passwd.f.name.e -side left
frame .passwd.f.pass
pack configure .passwd.f.pass -side top -fill x
# Frame qui va contenir le label "Type your password:" et une entrée pour le rentrer
label .passwd.f.pass.l -text {Type your password:}
pack configure .passwd.f.pass.l -side left
entry .passwd.f.pass.e -textvariable getPassword(passwd) -show "*"
pack configure .passwd.f.pass.e -side left
# L'option -show permet de masquer la véritable entrée, et de mettre une étoile à la place des
# caractères entrés
frame .passwd.f.buttons
pack configure .passwd.f.buttons -side top -fill x
# Frame qui va contenir les boutons Cancel et Ok
button .passwd.f.buttons.cancel -text Cancel -command {destroy .passwd}
pack configure .passwd.f.buttons.cancel -side left
button .passwd.f.buttons.ok -text Ok -command {set getPassword(result) 1;destroy .passwd}
pack configure .passwd.f.buttons.ok -side right
grab set .passwd
tkwait window .passwd
if {$getPassword(result)} {
 return [list $getPassword(name) $getPassword(passwd)]
} else {
 return ""
 }
}
```

V. Interfaçage avec d'autres langages

Tk est très puissant, malheureusement Tcl l'est moins, ses principaux points faibles sont:

- Sa vitesse, bien qu'améliorée à la version 8 par une semi-compilation du code
- Son manque de modularité
- Son absence de typage

C'est pour cela que lorsque l'on veut écrire une grosse application, il est indispensable de passer par un langage classique (C, Ada, et pourquoi pas ML...).

A. Interfacer C et Tcl/Tk

Lier dynamiquement un langage compilé (C en l'occurrence), et un langage interprété comme TCL/TK paraît étrange, mais le fait est que cela fonctionne très bien. Tcl/Tk est écrit en C et les bibliothèques C restent. La fonction permettant d'interpréter du script Tcl/Tk est elle aussi écrite en C et donc utilisable dans un programme C.

Le principe de programmation est le suivant:

1. Les différentes étapes

a) Script Tcl/Tk

Associer a chaque événement qui doit être récupéré par le programme C une variable (globale) TCL. Par exemple si le programme C doit être averti d'un click sur le bouton OK, le script TCL doit lier l'événement click souris sur OK avec une écriture dans une variable TCL.

b) Source C

- Associer le fichier TCL contenant l'interface graphique à un interpréteur logique du type Tcl_Interp.
- Définir des fonctions spécifiques au traitement des occurrences d'événements X à récupérer (click sur OK).
- Lier les variables TCL modifiées lors de l'arrivée d'un événement à la fonction de traitement adéquate.

2. Créer un interpréteur logique TCL

Un interpréteur logique (type *Tcl_interp*) est ce qui lie le source C au fichier TCL. La première chose à faire est de créer un interpréteur par la fonction :

*Tcl_Intepr * Tcl_CreateInterp()*.

L'interpréteur créé contient au retour de chaque fonction TCL appelée des informations sur le statut de la fonction: quel type d'erreur elle contient , etc... Tout ceci est dans un champs nommé *result* du type *Tcl_interp*.

Ensuite il faut initialiser l'interpréteur TCL par
*int Tcl_Init(Tcl_interp * interp)*
initialiser l'interpréteur pour qu'il gère TK par
int Tk_Init(Tcl_Interp interp)*
ces deux fonctions étant appelées successivement sur l'interpréteur renvoyé par *Tcl_CreateInterp*.

Il ne reste plus qu'à charger le fichier TCL/TK à interpréter à l'aide de la fonction
*int Tcl_EvalFile(Tcl_Interp * interp, char * Path)* où path est le chemin d'accès du fichier et interp le même interpréteur que précédemment.

Ca y est, le fichier est interprété en parallèle avec le programme C. Toutes ces manipulations étant assez ennuyeuses, elles sont regroupées dans une fonction que j'ai écrite.

Cette fonction est donnée en annexe, et pour les paresseux du clavier, elle peut être récupérée sur mon adresse http:
http://www.lisi.univ-poitiers.fr/members/grolleau
Elle est définie par *Tcl_Interp * InterpreterfichierTCL(char * Path)*
Elle renvoie l'interpréteur du fichier TCL donné dans *Path*, qui est interprété.

3. Définir des fonctions de traitements des événements TCL/TK

Il y a plusieurs moyens de récupérer les événement TCL/TK dans le source C. Cependant, il n'y a, à ma connaissance, qu'une seule méthode pour rendre ces événements récupérables par C dans le script TCL:

a) Rendre les événements TCL récupérables par C:

Imaginons que l'on veuille passer l'événement "Click sur OK" à C. Il suffit, dans le "binding" du bouton OK, d'exécuter la ligne:
***set** OK 1*.
Ainsi à chaque fois que l'utilisateur clique sur OK, la variable OK est accédée en écriture.

Maintenant le stratagème utilisé pour récupérer ce click dans le programme C est de tracer la variable *OK* du script TCL à l'aide de la fonction
*Tcl_TraceVar(Tcl_Interp * interp, char * VariableTCL, int flags, Tcl_VarTraceProc * Proc, ClientData data)*

- L'interpréteur est celui qui est lié au script TCL interprété, comme d'habitude.
- *VariableTCL* est le nom de la variable TCL à tracer.
- *flags* peut recevoir TCL_TRACE_WRITES, ainsi la fonction Proc passée en argument est appelée dès que la variable TCL tracée est accédée en écriture. Il y a d'autre flags possible mais là, se reporter à la documentation *TraceVar*.
- Pour le *ClientData* je passe toujours *(ClientData)NULL*

Les fonctions appelées lors du traçage d'une variable sont définies de la façon suivante:

*char * ma_fonction_de_traitement (ClientData clientData /* Ca, je n'en tiens jamais compte*/, Tcl_Interp interp, char * name1, char * name2, int flags)*

C'est TCL qui va se charger d'appeler la fonction de traitement C et d'en remplir les champs dès que la variable TCL tracée est accédée en écriture. Cette fonction peut utiliser des variables internes au programme C à l'intérieur de son corps exactement comme une fonction ordinaire.

- *interp* est l'interpréteur du fichier TCL contenant la variable tracée
- *name1* et *name2* permettent de constituer le nom de la variable TCL tracée. Lorsque celle-ci est une variable de type scalaire, *name1* contient son nom alors que *name2* ne sert à rien; par contre si elle est de type tableau, *name1* est le nom du tableau et *name2* l'indice du tableau. *name1* et *name2* peuvent permettre par exemple de lire la valeur de la variable TCL par la fonction *GetVar2*.

 Il est indispensable que les fonctions ainsi définies retournent une valeur. Pour ma part je termine toutes mes fonctions de traitement par *return NULL*.

Une fois que le fichier TCL est interprété, que toutes les variables TCL à tracer ont ainsi été liées à des fonctions C, il suffit de lancer la fonction *Tk_MainLoop()* qui donne la main à l'interface et qui se charge d'appeler les fonctions C de traitement liées à des variables tracées.

Pour un exemple simple, voir le fichier *exemple_C_Tcl.c* en annexe.

B. Note à l'attention des programmeurs ADA:

Puisqu'en ADA, à ma connaissance, il est impossible de passer un pointeur de fonction en paramètre d'une autre, il existe une autre méthode, un peu plus lourde, pour récupérer les événements TCL.

Elle est basée sur la même méthode de programmation du script TCL que précédemment (écriture d'une variable TCL à chaque événement à récupérer), mais là il faut modifier la valeur de cette variable à chaque occurrence d'événement. Le programme ADA se contentera de scruter les valeurs de ces variables dans la boucle principale du programme à l'aide de fonctions comme GetVar ou bien, si c'est possible, LinkVar (mais je ne sais pas si il est possible de lier des variables ADA aux variables TCL comme des variables C au programme TCL). Plus proprement cela peut être fait à l'aide d'une tâche Ada scrutant les événements Tcl/Tk et appelant éventuellement d'autres tâches pour réagir aux événements...
L'interfaçage des fonctions C de Tcl/Tk avec Ada a été faite par Jean-Claude Potier, et ça fonctionne très bien.

C. Note à l'attention des programmeurs autres que C et Ada

A priori, tout langage interfaçable avec C est interfaçable avec Tcl/Tk. Il suffit d'interfacer chaque fonction des librairies Tcl et Tk dans le langage, puis de les utiliser comme n'importe quelle autre fonction.

D. Compilation et édition des liens

Pour utiliser toutes les fonctions Tcl/Tk, il faut dans le source:
#include <tcl.h>
#include <tk.h>

Pour compiler, il faut donner l'endroit où le compilateur c peut trouver ces fichiers d'en-tête: **cc monfichier.c –Ichemin_de_tk.h_et_tcl.h** sur alienor, ils sont dans /home/local/include
Puis pour lier les objets il faut inclure les librairies libtcl.a et libtk.a avec les options -ltk et -ltcl. De plus si ces librairies ne sont pas dans les répertoires par défaut ajouter –Lchemin_des_librairies_tcl_et_tk, sur alienor c'est /home/local/lib.

 L'ordre -ltk -ltcl, tk avant tcl, est important lors de l'édition de liens.

VI. ANNEXES

Annexe 1: Fonction de chargement d'un fichier TCL/TK

tcl_tk.h

```
#ifndef _TCL_TK_H
#define _TCL_TK_H

#include <tk.h>
Tcl_Interp * InterpreterFichierTCL(char * Path);
/* IN    : Path chemin du fichier TCL                    */
/* RETURN: l'interpreteur qui s'occupe du fichier TCL    */

#endif
```

tcl_tk.c

```
/* Interprete un fichier TCL donne en parametre          */
Tcl_Interp * InterpreterFichierTCL(char * Path)
/* IN    : Path chemin du fichier TCL                    */
/* RETURN: l'interpreteur qui s'occupe du fichier TCL    */
{
   Tcl_Interp * interp;
   interp=Tcl_CreateInterp();
   /* Interpreteur TCL logique, a creer avant tout */
   /* Initialisations de l'interpreteur et de la fenetre */
   if (Tcl_Init(interp)!=TCL_OK)
   {
      fprintf(stderr,"Tcl_Init failed: %s\n",interp->result);
   }
   if (Tk_Init(interp)!=TCL_OK)
   {
      fprintf(stderr,"Tk_Init failed: %s\n",interp->result);
   }
   /* Lecture et interpretation du fichier TCL */
   if (Tcl_EvalFile(interp,Path)!=TCL_OK)
   {
      fprintf(stderr,"%s\n",interp->result);
      fprintf(stderr,"%s\n",Tcl_GetVar(interp,"errorInfo",TCL_GLOBAL_ONLY));
   }
   return interp;
}
```

Annexe 2: Exemple de source C utilisant un module TCL

```
/* exemple_C_Tcl.c                                          */
/* Fichier exemple d'interfacage TCL/TK avec C              */
/* doit etre compile avec les options :                     */
/* -I<repertoire de tk.h>  sur alienor /home/local/include  */
/* -L<repertoire de libtk.a> sur alienor  /home/local/lib   */
/* -lX11 -lm -ltk -ltcl                                     */
/* -o exemple                                               */
/* Exemple: cc exemple_C_Tcl.c -I/home/local/include -L/home/local/lib -lX11 -lm -ltk -ltcl -
o exemple*/

/* Le source Tcl contient une variable globale _TK_EXIT qui est accédée en écriture lorsque
l'utilisateur veut quitter */

#include "tcl_tk.h"

Tcl_Interp * interp;
/* Interpréteur Tcl */

char * _tk_exit(ClientData clientdata,Tcl_Interp inter,char * name1,char * name2,int flags)
/* Fonction appelée par Tcl lorsque la variable Tcl _TK_EXIT sera modifiée */
{
        char * v;
        int i;
        v=Tcl_GetVar2(inter,name1,name2,0);
        sscanf(v,"%d",&i);
        if (i) Tcl_GlobalEval(interp,"exit");
        return NULL;
        /* ne pas oublier de retourner NULL => Sinon cela plante*/
}

void main(int argc,char * argv[])
{
        if (argc<2)
        {
          fprintf(stderr,"interpret2 <fichier_tcl>\n");
          return;
        }
        InterpreterFichierTCL(argv[1]);
        Tcl_TraceVar(interp,"_TK_EXIT",TCL_TRACE_WRITES,_tk_exit,(ClientData)NULL
);
        /* c'est ici que l'on lie la fonction C avec la modification de la variable Tcl */
        /* Boucle obligatoire */
        Tk_MainLoop();
}
```

Annexe 3: Les fonctions C/TCL/TK les plus communément utilisées

char * Tcl_SetVar(Tcl_Interp * interp, char * varName, char * newValue, int flags)
DESCRIPTION: Modifie la valeur d'une variable TCL
RETOUR: si NULL il y a eu une erreur
IN: interp interpréteur donné par Tcl_CreateInterp()
varName: nom de la variable TCL à modifier
newValue: chaîne contenant la valeur que l'on veut affecter à la variable (donc pour les entiers utilisation de sprintf pour conversion d'entiers en chaînes)
flags: je conseille l'utilisation de TCL_GLOBAL_ONLY pour ce paramètre
OUT:
IN OUT:
REMARQUES: je crois que cette modification est prise exactement comme la ligne de commande *set varName newValue* et en particulier l'écriture dans une variable tracée va déclencher la fonction C associée à la variable modifiée. Attention donc à ne pas modifier de la sorte une variable tracée dans la fonction associée à son trace!!!
De plus si la variable n'existe pas elle est créée.

char * Tcl_GetVar(Tcl_Interp * interp, char * varName, int flags)
DESCRIPTION: Renvoie la valeur d'une variable TCL
RETOUR: si NULL il y a eu une erreur, une chaîne contenant la valeur de la variable TCL lue sinon.
IN: interp interpréteur donné par Tcl_CreateInterp()
varName: nom de la variable TCL dont on veut la valeur
flags: je conseille l'utilisation de TCL_GLOBAL_ONLY pour ce paramètre
OUT:
IN OUT:
REMARQUES:

char * Tcl_GetVar2(Tcl_Interp * interp, char * name1, char * name2, int flags)
DESCRIPTION: Même fonction que GetVar sauf qu'elle permet en plus d'accéder à des tableaux TCL. Ainsi si name1 contient quelque chose comme "montab()" et name 2 contient "4" la variable lue est montab(4). En fait, bien on utilise cette fonction souvent dans les fonctions définies pour récupérer les événements, car les fonctions sont appelées avec les paramètres name1 et name2 qui servent à l'appel de cette fonction, sans être obligé de connaître les détails de ses arguments.

int Tcl_LinkVar(Tcl_Interp * interp, char * varName, void * varCName, int type)
DESCRIPTION: Lie une variable C à une variable TCL, ainsi cette variable est partagée par C et TCL.
RETOUR: TCL_OK ou TCL_ERROR
IN: interp interpréteur donné par Tcl_CreateInterp()
varName: nom de la variable TCL à lier à varCName
varCName: nom de la variable C à lier à varName
type: TCL_LINK_INT pour les entiers
TCL_LINK_DOUBLE pour les double(flottants)

TCL_LINK_BOOLEAN: la variable C est un *int* mais la conversion 0→0 et autre→1 est faite pour la valeur de la variable TCL qui reste ainsi un booléen.

TCL_LINK_STRING pour les châines de caractères

combinaison éventuelle par un ou bit à bit avec TCL_LINK_READ_ONLY ainsi la variable ne peut être modifiée que par le programme C et est en lecture seule pour TCL.

OUT:

IN OUT:

REMARQUES: Attention, modifier la valeur d'une variable liée en C n'active pas la trace éventuelle d'une variable. i.e. si la variable C est modifiée et que la variable TCL liée est tracée en écriture, la fonction de traitement appelée en cas de modification n'est pas appelée.

int Tcl_GlobalEval(Tcl_Interp * interp, char * Commande)

DESCRIPTION: Exécute une ligne de commande TCL, très pratique pour modifier dynamiquement une interface graphique.

RETOUR: TCL_OK ou TCL_ERROR

IN: interp interpréteur donné par Tcl_CreateInterp()

Commande: une commande TCL/TK ordinnaire

OUT:

IN OUT:

REMARQUES: