Aide-mémoire pour ezTK

Version 2024-02 ***Christophe Schlick***

Ce document a pour objectif d'être un ***aide-mémoire*** présentant de manière synthétique, l'ensemble des fonctionnalités offertes par la bibliothèque **ezTK** *(prononcer `easy-téka`)* utilisable avec le langage de programmation **Python**. Comme son acronyme le laisse supposer, cette bibliothèque a été conçue pour permettre à des programmeurs débutants de réaliser ***très simplement et très rapidement*** des applications Python mettant en oeuvre des **interfaces graphiques** (***Graphical User Interface***), pilotées au clavier et à la souris, et fonctionnant de manière quasi-identique sous les plateformes Windows, MacOS et Linux.

Concrètement, la bibliothèque **ezTK** se présente comme une couche simplificatrice de la bibliothèque **tkinter** qui fait partie des modules disponibles en standard avec l'environnement de développement Python. La bibliothèque **tkinter** est elle-même une adaptation au langage Python d'une bibliothèque encore plus générale, appelée **Tk**, dont le développement a débuté il y a plus de 30 ans et qui a été adaptée pour plus d'une vingtaine de langages de programmation différents.

Comme c'est le cas pour tout aide-mémoire, ce document ne vous permettra pas d'apprendre à programmer avec **ezTK** mais vous permettra uniquement de retrouver les noms, les rôles et les paramètres des différentes fonctions fournies par le module, lorsque vous aurez un doute sur tel ou tel point de syntaxe ou de mise en oeuvre. En fait, la seule et unique manière efficace pour arriver à maîtriser n'importe quelle bibliothèque logicielle disponible pour n'importe quel langage de programmation consiste à ***lire, exécuter, comprendre et modifier des exemples de programmes mettant en oeuvre les fonctionnalités offertes par cette bibliothèque***.

C'est pour cela que ce document est indissociable d'une autre ressource qui vous est fournie dans le cadre de l'UE "Programmation et Applications Interactives", il s'agit des ***32 exemples de programmes*** disponibles dans la rubrique **Exemples de Niveau C** sur le site web du cours. Pour simplifier le téléchargement, l'ensemble des fichiers nécessaires pour tester tous ces exemples ont été regroupés dans une **archive ZIP**. Les codes source de tous ces exemples (numérotés de **1A** à **7F**) ont également été fusionnés dans un **fichier PDF** que vous pourrez soit imprimer, soit visualiser directement dans un onglet de votre navigateur. Il est ***fortement conseillé*** de lire et de tester ces exemples dans l'ordre indiqué, car les notions y sont présentées de manière progressive, et il est ***fortement déconseillé*** de passer à l'exemple suivant tant que vous n'avez pas compris l'intégralité des lignes de code de l'exemple courant. Voici la liste des 7 thématiques qui sont abordées successivement par ces 32 exemples :

- ✦ **C1A ... C1F** : création d'une fenêtre et organisation de son contenu
- ✦ **C2A ... C2D** : mise en oeuvre du mécanisme d'animation automatique
- ✦ **C3A ... C3C** : principe général de l'interaction homme-machine *(callbacks)*
- ✦ **C4A ... C4E** : mise en oeuvre des différentes widgets d'interaction
- ✦ **C5A ... C5D** : création de fenêtres multiples dans une application

- ✦ **C6A ... C6D** : gestion des interactions effectuées au clavier et à la souris
- ✦ **C7A ... C7F** : utilisation de la widget Canvas

A - Introduction

1 - Installation de la bibliothèque *ezTK*

La bibliothèque **ezTK** ne fait pas partie des modules disponibles en standard avec l'environnement de développement Python. Il faut donc que l'interpréteur Python sache comment trouver le code source de la bibliothèque pour pouvoir en utiliser les fonctionnalités. Les bibliothèques non standards nécessitent habituellement une procédure d'installation spécifique pour les faire fonctionner, mais dans le cas d'**ezTK** comme la bibliothèque se compose d'un seul et unique fichier, le processus est plus simple : ***il suffit de télécharger ce fichier, nommé ezTK.py, et de le stocker dans le dossier où se trouvent les fichiers des applications qui utilisent la bibliothèque.***

Ensuite, pour pouvoir utiliser la bibliothèque, il suffit d'importer l'intégralité des fonctionnalités avec la commande suivante :

```
from ezTK import *
```

2 - Liste des widgets

Quelque soit la bibliothèque utilisée, le principe général de développement d'une interface graphique consiste à créer une fenêtre principale et y ajouter un certain nombre d'éléments pour afficher des données et permettre à l'utilisateur d'interagir avec elles. Ces éléments s'appellent des **widgets**, terme obtenu par contraction de ***window gadgets***. On peut considérer les widgets comme des briques que l'on va combiner de manière très libre pour créer à la fois l'aspect graphique et l'aspect fonctionel des interfaces utlisateurs. La bibliothèque **ezTK** propose ainsi 12 types de widgets, qui se répartissent en trois grandes catégories :

- ✦ **Widgets d'organisation :** `Win, Frame`
- ✦ **Widgets d'information :** `Brick, Label, Listbox, Canvas`
- ✦ **Widgets d'interaction :** `Button, Scale, Spinbox, Entry, Menu, Dialog`

3 - Syntaxe générale

Pour chacun des 12 types de widgets disponibles dans **ezTK** est associée une ***fonction de création*** spécifique. Donc pour créer une widget d'un type donné, il suffit d'appeler la fonction correspondante et de lui fournir les paramètres de configuration. La syntaxe générale de ces appels de fonction est la suivante :

```
widgetName = WidgetType(master, propA=valA, propB=valB, ...)
```

✦ `widgetName` correspond au nom de la variable qui va stocker la référence de la widget créée par la fonction
(par convention, les noms des widgets commencent par une lettre minuscule)

✦ `WidgetType` correspond au type de widget à créer, parmi les 12 types listés ci-dessus
(par convention, les types de widgets commencent par une lettre majuscule)

✦ `master` correspond à la référence d'une widget d'organisation (de type `Win` ou `Frame`) déjà créée, dans laquelle on souhaite insérer la nouvelle widget.

✦ `prop=val` correspond aux couples **propriété = valeur** permettant de configurer l'aspect visuel et/ou fonctionnel de la widget.

Lors de la création d'une widget, seul le paramètre `master` est obligatoire, car il faut systématiquement préciser à quel endroit de la fenêtre, la nouvelle widget est insérée. Seule les fenêtres autonomes de l'application (appelées *master windows*) ne possède pas de paramètre obligatoire `master` (cf. *Exemple C6A*).

A l'inverse, tous les couples `prop=val` sont optionnels dans l'appel de la fonction de création de la widget. Cela est lié au fait que ***chaque propriété possède une valeur par défaut***, ainsi toute propriété qui n'est pas passée en paramètre va automatiquement être initialisée à sa valeur par défaut. La ***section E*** à la fin de ce document va détailler le rôle et la valeur par défaut pour chacune des propriétés définies par **ezTK**, en les listant par ordre alphabétique pour permettre de les retrouver facilement. Mais évidemment, toutes les propriétés ne s'appliquent à toutes les widgets, c'est pour cela qu'à chaque type de widget décrit dans les ***sections B, C et D***, on fournira la liste des propriétés utilisables pour ce type de widget.

B - Widgets d'organisation

Les **widgets d'organisation** (également appelées ***conteneurs***) sont utilisées pour définir la mise en page des widgets présentes dans la fenêtre. Le principe général consiste à organiser les widgets de manière hiérarchique, en associant une zone rectangulaire de la fenêtre pour chacun des niveaux hiérarchiques.

1 - Win

`Win` est une widget d'organisation qui permet de créer une fenêtre autonome sur l'écran (cf. *Exemples C1A, C1B, C1C, C6A*)

Propriétés : `title, grow, bg, fg, font, width, height, op, ip, flow, fold, relief, border, click, key, inout, move`

Méthodes : `loop(), after()` ,

2 - Frame

`Frame` est une widget d'organisation qui permet de définir une zone rectangulaire dans laquelle seront regroupées d'autres widgets. Une `Frame` peut contenir un nombre quelconque de widgets,

y compris d'autre `Frame` ce qui permet une organisation hiérarchique de la mise en page de la fenêtre. Les propriétés `flow` et `fold` sont les plus importantes d'une widget `Frame` car elles contrôlent le positionnement géométrique des widgets insérées dans le conteneur, ainsi que la manière dont les widgets sont numérotées automatiquement. Ce fonctionnement est détaillé dans la section ***Mise en page***, à la fin de ce document (cf. *Exemples C1D, C1E, C1F*)

Propriétés : `grow, bg, fg, font, width, height, op, ip, flow, fold, relief, border`

C - Widgets d'information

Les **widgets d'information** sont utilisées pour visualiser les données destinées à l'utilisateur (valeurs numériques, messages d'information, blocs de texte, images, couleurs...)

1 - Brick

`Brick` est une widget permettant d'afficher, soit une zone rectangulaire de couleur unique définie par la propriété `bg` , soit une image définie par la propriété `image` (cf. *Exemples C1A, C1B, C2A, C2B, C2C*)

Propriétés : `grow, bg, image, width, height, relief, border`

2 - Label

`Label` est une widget permettant d'afficher une chaîne de caractères définie par la propriété `text` , associée à une police définie par la propriété `font` (cf. *Exemples C1C, C2D, C3A, C3B, C3C*)

Propriétés : `grow, bg, fg, text, font, width, height, relief, border`

3 - Listbox

`Listbox` fournit une widget permettant d'afficher une liste de chaîne de caractères (une par ligne) dans une zone de texte. Le contenu d'une `Listbox` se gère comme le contenu d'une liste Python standard (cf. *Exemple C4E*)

Propriétés : `grow, bg, fg, image, width, height, scroll, relief, border`

4 - Canvas

`Canvas` fournit une widget très générale permettant de créer une zone d'affichage rectangulaire qui peut contenir des éléments variés (textes, images, tracés géométriques) qui peuvent même être animées (cf. *Exemples C7A, C7B, C7C, C7D, C7E*)

Propriétés : `grow, bg, fg, width, height, scroll, relief, border`

Méthodes : `create_rectangle, create_oval, create_line, create_arc,`

`create_polygon, create_image, create_text`

D - Widgets d'interaction

Les **widgets d'interaction** sont utilisées pour récupérer les actions de l'utilisateur au clavier et à la souris

1 - Button

`Button` fournit une widget similaire à une `Brick` (affichage d'une image) ou un `Label` (affichage de texte) mais qui permet à l'utilisateur de lancer une action, définie par la propriété `command`, lorsqu'il clique sur la zone rectangulaire associée à la widget (cf. *Exemples C3A, C3B, C3C*)

Propriétés : `grow, bg, fg, text, font, image, width, height, relief, border, command`

2 - Scale

`Scale` fournit une widget qui permet à l'utilisateur de choisir une valeur numérique (entière ou réelle) entre 2 bornes prédéfinies. Il se présente sous la forme d'un potentiomètre horizontal ou vertical (cf. *Exemple C4A*)

Propriétés : `grow, scale, troughcolor, width, height, orient, relief, border, state, command`

3 - Spinbox

`Spinbox` : fournit une widget qui permet à l'utilisateur de choisir un élément dans une liste prédéfinie contenant des valeurs numériques ou des chaînes de caractères (cf. *Exemple C4B*)

Propriétés : `grow, bg, fg, text, font, width, height, relief, border, state, command`

4 - Entry

`Entry` : fournit une widget qui permet à l'utilisateur de saisir un mot ou une ligne de texte au clavier (cf. *Exemple C4C*)

Propriétés : `grow, bg, fg, text, font, width, height, relief, border, state, command`

5 - Menu

`Menu` : fournit une widget qui permet de créer une barre de menus déroulants pour la fenêtre (cf. *Exemple C4D*)

Propriétés : `grow, bg, fg, text, font, width, height, relief, border`

Méthodes : `add_command, add_checkbutton, add_radiobutton, add_cascade`

6 - Dialog

`Dialog` : fournit une widget qui permet de créer différents types de fenêtres de dialogue, qui viennent s'afficher temporairement au-dessous de la fenêtre principale de l'application pour fournir des notifications ou poser des questions à l'utilisateur (cf. *Exemple C5D*)

Propriétés : `title, mode, message`

E - Propriétés des widgets

Cette section va lister, par ordre alphabétique, l'ensemble des propriétés utilisables pour l'ensemble des widgets fournies par **ezTK**. Il faudra se référer aux ***sections B, C et D*** ci-dessus, pour avoir la liste précise des propriétés qui s'appliquent effectivement pour chacun des 12 types de widgets.

Pour chaque propriété, la liste indique également la ***valeur par défaut*** (placée juste après le signe `=`) qui sera automatiquement utilisée si l'utilisateur ne fournit pas de valeur spécifique pour cette propriété :

✦ `anchor='C'` :
Une chaîne de caractères qui définit la manière dont le contenu d'une widget (texte ou image) est ancrée ***(anchor)*** par rapport à la zone rectangulaire qu'occupe la widget à l'écran. Neuf valeurs sont autorisées : `'C' 'N' 'NE' 'E' 'SE' 'S' 'SW' 'W' 'NW'` sachant que les 8 dernières correspondent aux 8 directions cardinales d'une boussole, et la valeur par défaut `'C'` signifie que le contenu est placé au centre de la widget

✦ `border=0` :
Une valeur entière qui définit la largeur de la bordure entourant la widget, exprimée en ***nombre de pixels***.
La bordure sera transparente si `relief='flat'` ou noire si `relief='solid'`

✦ `bg='white'` :
Une chaîne de caractères qui définit la couleur du fond ***(background color)*** de la widget.
Le formalisme utilisé est détaillé dans la section ***Couleurs*** ci-dessous

✦ `command=None` :
Le nom d'une fonction qui définit l'action ***(callback)*** associée à une widget d'interaction.
Il existe 3 manière de définir la fonction d'action (cf. *Exemples C3A, C3B et C3C*) :

- Si la fonction est appelée sans fournir de paramètres, on utilise simplement le nom de la fonction :

```
command=func
```

- Si la fonction est appelée avec des paramètres constants `valA,valB,...`, on

utilise la forme courte de l'instruction `lambda` pour fournir les paramètres transmis à la fonction :

```
command=lambda:func(valA,valB,...)
```

● Si la fonction est appelée avec des paramètres variables `varA,varB,...` , on utilise la forme longue de l'instruction `lambda` pour fournir les paramètres transmis à la fonction :

```
command=lambda a=varA,b=varB,...:func(a,b,...)
```

✦ `fg='black'` :
Une chaîne de caractères qui définit la couleur du texte ou du tracé ***(foreground color)*** de la widget.
Le formalisme utilisé est détaillé dans la section ***Couleurs*** ci-dessous

✦ `fold=None` :
Une valeur entière qui définit le nombre de widgets qui peuvent être insérées séquentiellement dans un conteneur avant repliement ***(fold)***, c'est-à-dire avant de passer à la ligne suivante (si la direction d'insertion est horizontale) ou à la colonne suivante (si la direction d'insertion est verticale)
Cette propriété n'existe que pour les widgets d'organisation `Win` et `Frame` qui sont les seules à pouvoir contenir d'autres widgets (cf. *Exemples C1B, C1C, C1D*)

✦ `flow='SE'` :
Une chaîne de caractère qui définit la direction du flux ***(flow)*** utilisée pour insérer les widgets dans un conteneur.
Cette propriété est fortement liée à la propriété `fold` décrite ci-dessus, et comme elle, n'existe que pour les widgets d'organisation `Win` et `Frame` qui sont les seules à pouvoir contenir d'autres widgets (cf. *Exemples C1B, C1C, C1D*) :

● Si `fold=None` (= pas de repliement), le flux est appelé ***monodirectionnel*** et est défini par une seule direction cardinale `'N','E','S','W'` indiquant la direction de remplissage du conteneur

● Si `fold=n` (= repliement au bout de **n** widgets), le flux est appelé ***bidirectionnel*** et utilise deux directions cardinales (primaire et secondaire), qui doivent nécessairement être orthogonales (l'une verticale, l'autre horizontale). La direction primaire correspond à la direction d'insertion des widgets dans le conteneur, alors que la direction secondaire correspond à celle du repliement

✦ `font='Arial 12'` :
Une chaîne de caractères qui définit la police de caractères ***(font)*** utilisé pour le texte affiché dans la widget.
Le formalisme utilisé est détaillé dans la section ***Polices*** ci-dessous

✦ `grow=True` :
Un booléen qui définit si la widget est autorisée à grandir ***(grow)*** ou pas, lorsque l'utilisateur redimensionne la fenêtre de l'application

✦ `height=None` :
Un entier qui définit la hauteur de la widget à l'écran. Par défaut, la hauteur est calculée automatiquement en fonction du contenu stocké dans la widget. Si une taille est fournie par l'utilisateur, elle s'exprime en ***nombre de pixels***, sauf si la widget contient du texte, auquel cas, la largeur s'exprime en ***nombre de lignes de texte*** (dans ce cas, la hauteur finale dépendra donc de la taille de la police utilisée).

✦ `image=None` :
Une variable de type `Image` ou `ImageGrid` qui définit les images qui peuvent être affichée dans la widget (cf. *Exemples C2C, C7C, C7E*).
Le formalisme utilisé est détaillé dans la section ***Images*** ci-dessous

✦ `ip=0` :
Une valeur entière qui définit la marge interne ***(ip = inner padding)*** de la widget, exprimée en ***nombre de pixels***. Cela correspond à la distance entre la bordure et le contenu (texte ou image) de la widget; cette distance est identique pour les 4 directions

✦ `op=0` :
Une valeur entière qui définit la marge externe ***(op = outer padding)*** de la widget, exprimée en ***nombre de pixels***. Cela correspond à l'écart entre la bordure et la bordure des widgets voisines; à nouveau cette distance est identique pour les 4 directions

✦ `relief='flat'` :
Une chaîne de caractères qui définit la décoration extérieure utilisée pour la widget. Six valeurs sont autorisées : `'flat' 'solid' 'raised' 'sunken' 'groove' 'ridge'`, sachant que les quatre dernières (qui utilisent un effet de trompe-l'oeil pour donner une impression de relief) ne sont plus vraiment utilisés par le look **'Flat Design'** des interfaces graphiques actuelles

✦ `scale=(0,100,1)` :
Un triplet qui définit les paramètres de variation d'une widget `Scale` : valeur minimale, valeur maximal, ainsi que le pas d'incrémentation. Si le troisième paramètre n'est pas fourni, il est automatiquement positionné à 1 (cf. *Exemple C4A*)

✦ `state=0` :
Un entier qui définit l'état de départ pour une widget multi-états (cf. la section ***Multi-états*** ci-dessous)

✦ `text=None` :
Une chaîne de caractères qui définit le texte affiché dans la widget (comme pour toute chaîne de caractère en Python, l'insertion du caractère `\n` va provoquer un retour à la ligne)

✦ `values=()` :
Un tuple de données arbitraires qui définit les différentes valeurs qui peuvent apparaître dans

une widget `Spinbox` (cf. *Exemple C4B*)

✦ `width=None` :
Un entier qui définit la largeur de la widget à l'écran. Par défaut, la largeur est calculée automatiquement en fonction du contenu stocké dans la widget. Si une taille est fournie par l'utilisateur, elle s'exprime en ***nombre de pixels***, sauf si la widget contient du texte, auquel cas, la largeur s'exprime en ***nombre de caractères*** (dans ce cas, la largeur finale dépendra donc de la taille de la police utilisée).

F - Eléments complémentaires

1 - Couleurs

Dans la plupart des applications logicielles, il existe deux manières de définir des couleurs :

✦ soit on utilise une ***couleur prédéfinie*** qui sera identifiée par une chaîne de caractères correspondant au nom de la couleur (en anglais). Les couleurs les plus classiques **black**, **white**, **blue**... sont évidemment présentes, mais vous pouvez utiliser près de 500 noms de couleurs standardisées dont la plupart sont regroupés sur **cette palette**

✦ soit on utilise une ***couleur personnalisée*** qui sera identifiée par une chaîne de caractères correspondant à la valeur des trois couleurs primaires exprimée en notation hexadécimale (= base 16).

Dans le cas des couleurs personnalisées, deux formats sont possibles pour spécifier les valeurs :

✦ Le format long `'#RRGGBB'` où `RR`, `GG` et `BB` représentent respectivement les valeurs des couleurs primaires ***(R = red, G = green, B = blue)*** définies en notation hexadécimale entre les valeurs `00` (= 0 en décimal) et `FF` (= 255 en décimal).
Ce formalisme permet ainsi de créer 256x256x256 soit environ ***16.8 millions de couleurs différentes***. Il existe des **outils dédiés** sur le web qui permettent de trouver interactivement les valeurs numériques des 3 couleurs primaires, pour obtenir n'importe quelle couleur souhaitée.

✦ Le format court `'#RGB'` est simplement un raccourci d'écriture pour les cas où les 2 chiffres hexadécimaux d'une même composante sont identiques : par exemple, `#5AF` est équivalent à `#55AAFF`.
Le format court permet de créer 16x16x16 soit ***4096 couleurs différentes*** ce qui est largement suffisant pour définir les couleurs d'une interface graphique

2 - Polices

En typographie, une police de caractères est définie par trois propriétés :

✦ un **nom**, qui caractérise la forme de base des caractères : par exemple **Arial**, **Courier**, **Times**...

✦ une **taille**, qui caractérise la hauteur des caractères : généralement une valeur comprise entre 4 et 120 points typographiques

✦ des **enrichissements**, qui caractérisent les modifications de la forme de base : par exemple **bold**, **italics**, **condensed**, **light**...

Comme pour les couleurs, le choix des polices de caractères utilisées par **ezTK** s'effectue par une chaîne de caractères avec deux formats possibles :

✦ Le format court `'nom taille'` qui ne définit que la forme de base et la taille :
par exemple **'Arial 12'** ou **'Courier 16'**

✦ Le format long `'nom taille enrichA enrichB...'` qui rajoute un nombre quelconque d'enrichissements (tous séparés par des espaces) :
par exemple **'Times 18 bold italics'** ou **'Verdana Pro Semibold 14 slanted'**

Il faut noter que les polices de caractères installées par défaut sur Windows, MacOS et Linux sont différentes. Sur chaque plateforme, il existe des tables de conversion sur chaque plateforme permettant de trouver une police similaire à une police demandée, mais cela ne fonctionne que pour les polices les plus classiques.

3 - Images

Une interface graphique utilise généralement de nombreuses images, principalement des **icônes** et des **sprites**, chacune correspondant à un élément visuel différent qui sera inséré dans le fenêtre de l'application. Les images manipulées par **ezTK** sont toujours stockées, au départ, dans des fichiers utilisant l'un des deux formats standards : le format **GIF** ou le format **PNG**. Pour charger une image depuis un fichier, on utilise la commande suivante :

```
imageName = Image(fileName)
```

✦ `fileName` : une chaîne de caractères qui définit le nom du fichier que la fonction `Image` va lire sur le disque et convertir en une grille de pixels stockée en mémoire

✦ `imageName` correspond au nom de la variable qui va stocker la référence de cette grille de pixels, et qui pourra être utilisée comme valeur pour la propriété `image` lors de la création d'une widget

Il arrive assez fréquemment que de nombreuses images utilisées par une application soient toutes de la même taille (par exemple, les différentes icônes d'une barre d'outils, ou les sprites identifiant les différents personnages dans un jeu vidéo 2D). Dans ce type de configuration, pour éviter de devoir créer un fichier individuel pour chacune des images, **ezTK** offre la possibilité de regrouper un nombre arbitrare d'images de taille identique dans un fichier GIF ou PNG unique, sous la forme d'une ***grille d'images***. Pour charger une image depuis un fichier, on utilise la commande suivante :

```
imageTuple = ImageGrid(fileName, rows, cols)
```

✦ `fileName` : une chaîne de caractères qui définit le nom du fichier que la fonction `ImageGrid` va lire sur le disque, extraire chacune des images de la grille d'images sous la forme d'une grille de pixels et stocker toutes les images dans un tuple

✦ `rows=None` : un entier qui définit le nombre de lignes de la grille d'images stockée dans le fichier, et qui va permettre de calculer la largeur de chacune des images présente dans la grille

✦ `cols=None` : un entier qui définit le nombre de colonnes de la grille d'images stockée dans le fichier, et qui va permettre de calculer la largeur de chacune des images présente dans la grille

✦ `imageTuple` correspond au nom de la variable qui va stocker le tuple contenant les différentes images

Note : Si l'utilisateur ne fournit qu'***une seule des deux valeurs*** `rows` ou `cols`, la fonction `ImageGrid` va supposer que les images sont de formes carrées et va automatiquement calculer l'autre valeur en fonction de la taille de l'image stockée dans le fichier. Et si l'utilisateur ne fournit ***aucune des deux valeurs***, la fonction va supposer que la grille ne contient qu'une seule ligne ou une seule colonne d'images carrées et va calculer la taille des images de la grille en fonction de la plus petite des 2 dimensions de l'image

4 - Multi-états

Lorsqu'on définit une widget **en utilisant un tuple pour définir une propriété** qui normalement attend une valeur unique, cette widget est automatiquement transformée en ***widget multi-états***. Les différentes valeurs fournies par le tuple sont numérotés (en commençant par 0) et correspondent aux différents **états** de la widget. Par exemple :

```
colors = ('red','green','blue')
brick = Brick(win, bg=colors)
```

permet de définir une widget `Brick` qui aura 3 couleurs possibles (rouge, vert et bleu). Lorsque la widget est créée, l'état par défaut est l'état 0 (donc ici, cela correspond à une couleur 'rouge')

Pour changer l'état d'une widget multi-état au cours de l'exécution du programme, il suffit de modifier la valeur de la propriété `state` associé à cette widget. Par exemple, avec l'exemple précédent, la commande `brick.state = 1` va faire passer l'état de la widget de 0 à 1, et donc automatiquement changer sa couleur de rouge à vert.

La propriété `states` permet de connaître de nombre total d'états pour une widget donnée, même si cette information est rarement utile. En particulier, il faut noter que si l'utilisateur affecte une valeur `state` qui est supérieure au nombre total d'états, **ezTK** va automatiquement appliquer un modulo `states` pour garantir que la widget reste toujours dans un état cohérent par rapport à sa définition initiale.

Les propriétés des widgets les plus utilisées pour définir des widgets multi-états sont `bg`, `fg`, `text` et `image` qui permettent de changer les couleurs, le texte ou l'image en fonction des actions de l'utilisateur (cf. *Exemples C2B, C2C, C2D, C6B, C6D*)

5 - Mise en page

La mise en page d'une interface est le processus qui permet de ***placer les différentes widgets les unes par rapport aux autres*** sur l'espace rectangulaire correspondant à la fenêtre de l'application. Sachant que les écrans des postes informatiques sont de résolutions variables d'une machine à l'autre, et que l'utilisateur va pouvoir, à tout moment, modifier la taille des fenêtres pour les applications en cours d'exécution, spécifier les coordonnées exactes pour chaque widget, est un processus beaucoup trop rigide. Au lieu de cela, la mise en page va consister pour le développeur à ***spécifier des règles générales pour positionner les widgets***, puis **ezTK** va utiliser un algorithme, appelée ***packer***, qui va calculer automatiquement la taille et la position de chaque widget en fonction de ces règles et de la taille courante de la fenêtre de l'application.

Les règles de mise en page sont principalement définies dans les widgets d'organisation `Win` et `Frame` avec les propriétés `flow` et `fold`, détaillées dans la ***section E*** ci-dessus. Néanmoins, toutes les autres widgets disposent d'une propriété `grow` qui permet de contrôler si une widget est autorisée à s'aggrandir (c'est le cas par défaut) ou non, lorsque l'utilisateur redimensionne la fenêtre de l'application. Ces trois propriétés, combinées à l'utilisation d'un organisation hiérarchique avec des widget `Frame` contenant d'autres widgets `Frame` permet de créer, très simplement, des mises en pages complexes avec plusieurs dizaines de widgets dans une même fenêtre qui vont se redimensionner et se repositionner harmonieusement, quelle que soit la taille courante de la fenêtre principale. Les *Exemples C1A à C1F* montrent, à travers une série d'exemples de plus en plus complexes, comment ce processus peut être mis en place pour obtenir des mises en page très diverses.

Christophe Schlick