1. Introduction

La possibilité de programmer des nouveaux objets, des outils ou des applications directement dans *ForthCAD* a été introduite dés les premières versions du logiciel.

A l'époque (1989-90), le langage ***Forth***[1] était l'un des rares langages de haut niveau qui offrait à la fois la compacité et les performances nécessaires.

L'interpréteur ***Lua***[2], dont le langage est à la fois plus simple et plus expressif, a été récemment ajouté, à partir de la version 62.00 de *ForthCAD* (Début 2014).

La suite de cet article -assez technique- s'adresse au lecteur qui possède les connaissances de bases d'un langage de programmation. Nous décrivons ici l'utilisation de *Lua* dans le but de personnaliser le logiciel *ForthCAD*.

Ressources Lua

La source de référence sur *Lua* se trouve sur le site Internet des auteurs :

- http://www.lua.org.

Le manuel « ***Programming in Lua*** » *de Roberto Ierusalimschy*, expose de façon progressive, claire et complète la programmation en *Lua*. Il est disponible en format papier ou sous forme numérique au format « *ebook* ».

L'API *Lua* propre à *ForthCAD* est disponible sous forme HTML à l'adresse Internet suivante :

- http://forthcad.com/fcad_lua_api.html

Contexte d'exécution

Un programme *Lua* peut être exécuté dans plusieurs contextes :

Au lancement du programme :

Si un fichier « .lua » est spécifié sur la ligne de commande, il est compilé et exécuté en premier[3].

A la création d'une fenêtre 2D, 3D ou texte :

Lors de création d'une première fenêtre interne, *ForthCAD* exécute le fichier « forthcad.lua », afin de configurer la barre d'outils verticale située à gauche. De ce fait, ce fichier est toujours exécuté au moins une fois à la mise en route du programme.

Le fichier « forthcad.lua » doit présenter une structure particulière. Il défini les boutons, bitmap, info bulles, raccourcis-clavier, l'activation et la désactivation des outils selon le contexte ainsi que le code correspondant à chaque bouton.

Lors d'un glisser-déposer « Drag & drop » :

Glisser un fichier d'extension « .lua » sur la fenêtre de *ForthCAD* exécute automatiquement le contenu du fichier.

Via entrées de commandes dans la console :

La console de commande, accessible via le menu « **Fenêtre / Console** », permet d'exécuter des instructions de programme.

A l'exécution des programmes, la console peut afficher divers messages, dont les messages d'erreurs.

La console peut aussi être utilisée comme une calculatrice, en introduisant le signe égal « = » suivi d'une expression arithmétique. (Voir § suivant).

Via un message externe de compilation :

Le programme ***fcompile.exe***, situé dans le même répertoire que *ForthCAD*, permet d'envoyer une requête d'interprétation d'un fichier *Lua* à une instance ouverte de *ForthCAD*.

Dans la console de *Windows*, la commande :

```
fcompile fichier.lua
```

...interprète le fichier source donné en paramètre, dans cet exemple, « fichier.lua ».

fcompile.exe permet aussi d'exécuter du code à partir d'un éditeur externe. Les messages d'erreurs sont retournés vers l'éditeur, qui peut ainsi mettre en exergue la ligne d'erreur.

Lors de l'utilisation d'un objet graphique Lua :

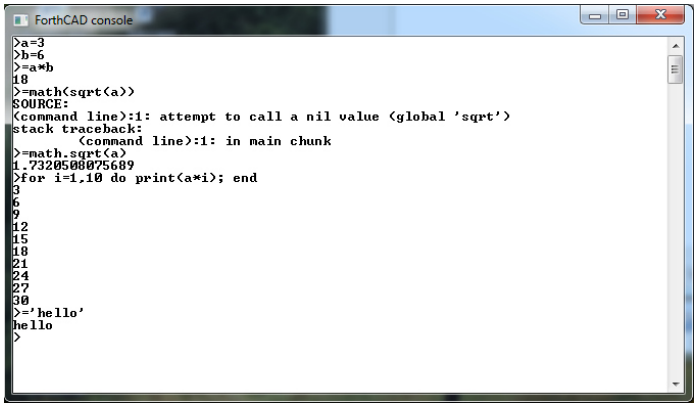
Lua permet de piloter des objets « intelligents » (portes, fenêtres, carrelages, objets paramétriques, etc.).

Il est également possible de créer des groupements spéciaux d'objets standards, et d'en gérer le contenu et les dimensions.

Le programme correspondant à ces objets est exécuté à la création, la modification, la sélection, et lorsque des transformations d'échelles sont appliquées à ces objets.

Console de commande

Dans *ForthCAD*[4], ouvrir la console de commande à partir du menu « **Fenêtre / Console** ».



Dans la console, introduire les commandes telles qu'elles apparaissent ci-dessus à droite du signe « **>** », puis presser **Enter**.

Lorsque la touche **Enter** est pressée, le texte de la ligne de commande est envoyé vers une instance de *ForthCAD* ouverte, pour y être interprété. Le résultat est ensuite retourné depuis *ForthCAD* vers la console.

Exemples

Pour tester la console, entrer les commandes suivantes :

```
print "hello"
```

[1] https://fr.wikipedia.org/wiki/Forth_(langage)

[2] https://fr.wikipedia.org/wiki/Lua

[3] Le nom du fichier source doit être indiqué entre guillemets.

[4] Les composants utilisés dans cet article supposent l'utilisation de la version 63.04 de ForthCAD.

La fonction « **print** » de *Lua* sert à afficher des valeurs dans la console de *ForthCAD*.

```
View():pushPoint(0,0)
View():pushPoint(100,0)
View():pushPoint(100,50)
```

Ces instructions ajoutent trois points de constructions dans la fenêtre courante de *ForthCAD* (2D ou 3D).

Pour afficher le nombre de points :

```
=View():getPointCount()
```

Pour créer un polygone à l'aide des points :

```
View():sendCmdMsg"polygon"
```

Cet usage de la console est évidemment fastidieux. La console n'est normalement utilisée que pour contrôler l'exécution d'un programme.

Utilisation d'un éditeur texte

Avant aller plus loin, l'utilisation d'un bon éditeur est nécessaire. Tout éditeur de texte capable de lancer une application externe peut convenir (*TextPad*, *Notepad++*, *Visual-Studio, ...*).

Un éditeur de texte préconfiguré pour *ForthCAD* est disponible sur notre site Internet.

Installation de l'éditeur

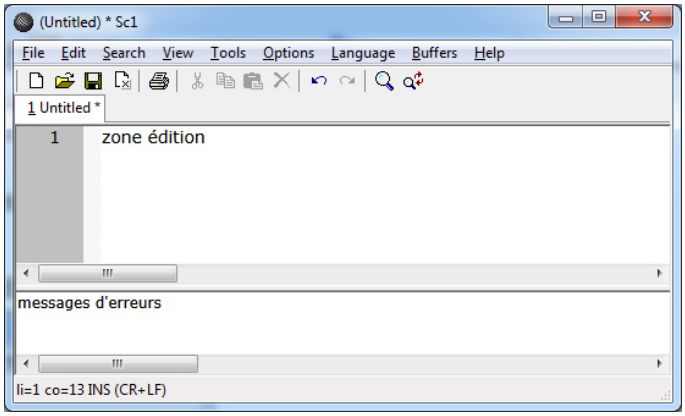
• Télécharger le fichier **SciTE.zip** à partir de la section « Téléchargement » du site http://forthcad.com.

• Extraire les fichiers de **SciTE.zip** dans le répertoire de travail de *ForthCAD*. L'extraction crée un sous-dossier nommé « **SciTE** ».

• Créer un raccourci pour faciliter le lancement du programme **« Sc353.exe »**.

Description de l'éditeur

Cet éditeur est une version allégée et préconfigurée de ***SciTE***[5].



Au dessus se trouve la zone d'édition. La colorisation syntaxique sera effectuée sur les fichiers suffixés « **.lua** ».

En dessous se trouve la zone d'affichage des messages d'erreurs. Cliquer un message d'erreur déplace le curseur sur la ligne en erreur dans la zone d'édition.

• Touche **F5** : Dans le menu « **Tools** », l'option « **Go F5** » exécute le programme dans ForthCAD (Qui doit donc être préalablement ouvert).

• Touche **Ctrl+F1** : « **Tools / ForthCAD API Help** » affiche l'aide relative à l'API *Lua* propre à *ForthCAD*. Ce document est chargé depuis Internet.

• Touche **F1** : affiche l'aide relative au langage *Lua*.

• L'option du menu « **Help / Sc1 help** » : Affiche de l'aide sur l'éditeur de texte.

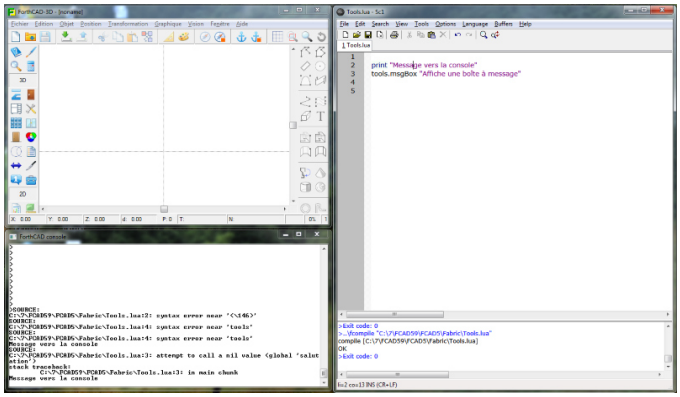
Des informations supplémentaires sont disponibles sur le site Internet[3] de *SciTE*.

Test de l'éditeur

• Lancer *ForthCAD* et ouvrir la console de commandes.

• Ouvrir l'éditeur *SciTE*.

• Disposer approximativement les 3 fenêtres comme sur l'image ci-après.



• Dans l'éditeur, frapper ou copier/coller le texte suivant :

```
print "Message vers la console"
tools.msgBox "Affiche une boîte à message"
```

• Enregistrer ce programme dans le fichier :

`FCAD5\Fabric\Tools.lua`

Où « FCAD5 » est le répertoire de travail de *ForthCAD*.

• Presser **F5** pour exécuter ce programme.

Si tout s'est bien passé, une boîte de dialogue est ouverte dans *ForthCAD*. Le message est aussi affiché dans la console de commande.

• Fermer la boîte de dialogue dans *ForthCAD*. Dans *SciTE*, modifier le texte comme suit :

```
print "Message vers la console"
salut()
tools.msgBox "Affiche une boîte à message"
```

• Presser **F5** pour exécuter ce programme.

Le message d'erreur...

`attempt to call a nil value (global 'salut')`

... est affiché à la fois dans la console et dans la fenêtre de sortie de *SciTE*. Cliquer sur ce message positionnera le curseur sur la ligne d'erreur.

• Corriger l'erreur et presser **F5** pour vérifier.

Test de la barre d'outils

• Dans *ForthCAD*, cliquer le dernier bouton  de la barre d'outils située à gauche.

Nous obtenons le même résultat que précédemment. La raison en est que ce bouton -normalement inutilisé- est relié au contenu du fichier proposé plus haut :

`FCAD5\Fabric\Tools.lua`

La mise en relation de ce bouton et de ce fichier à été prédéfinie dans le fichier d'initialisation « **forthcad.lua** ».

[5] Voir http://www.scintilla.org/SciTE.html

Fabriquer une échelle

L'exemple qui suit, très simple, permettra d'introduire plusieurs concepts de bases.

Il s'agit de créer une échelle dont toutes les dimensions sont fixées, à l'exception de sa hauteur.

Remplacer le programme de « **Tools.lua** » par le suivant :

```
local v = View()

-- (1) Vérifier contexte : (3D + 1 point)

if v:sn()~=3 or v:getPointCount()~=1 then
   tools.msgBox "Entrer 1 point en 3D"
   return -- termine le traitement
end

-- (2) Dimensions fixées

local P  = v:getPoint(1) -- position échelle
local de = 27   -- distance entre axes échelons
local em = 3.5  -- épaisseur des montants
local pm = 7    -- profondeur des montants
local re = 1.75 -- rayon des échelons
local w  = 80   -- largeur de l'échelle

-- (3) Entrer la hauteur désirée

local h, ok = tools.inputBox("Hauteur ?", 0)
if not ok then return; end -- Opération annulée

-- (4) Vérifier hauteur suffisante

if h < 3*de then
   tools.msgBox "Hauteur échelle trop petite"
   return
end

-- (5) Construire les montants

local montant={} -- tableau de montants
local echelon={} -- tableau d'échelons
montant[1]=Atom("BOX3D", P, Vec3(em, h, pm))
montant[2]=Atom("BOX3D",
                Vec3(P.x+w-em, P.y, P.z),
                Vec3(em, h, pm))

-- (6) Construire les échelons

local N = h//de -- nombre entier d'échelons.
de = h/(N+1)    -- recalibrer espace échelons
P.x = P.x+em; P.z = P.z+pm/2

local E = Vec3(P.x+w-2*em, P.y, P.z)

for i=1,N do
   P.y = P.y+de
   E.y = P.y
   echelon[i]=Atom("CYLINDER3D", P, E, re)
end

-- (7) Grouper { {montants} + {échelons} }

local group = Atom("GROUP",
                   { Atom("GROUP", montant),
                     Atom("GROUP", echelon) });

-- (8) Insérer l'objet dans la vue courante

v:addAtom(group)
```

Description du programme

Cette description suppose une maîtrise suffisante du langage *Lua*[6] et quelques notions de calcul vectoriel.

```
local v = View()
```

« **View** » est une classe prédéfinie de ForthCAD. Elle permet de gérer la vue courante. (Consulter L'API *Lua* propre à *ForthCAD* et voir plus loin dans ce document)

La variable « **v** » est un objet de type « **View** » qui permet d'obtenir des informations sur la fenêtre courante : Le type de fenêtre, le nombre de points, les objets, etc.

```
-- (1) Vérifier contexte : (3D + 1 point)

if v:sn()~=3 or v:getPointCount()~=1 then
   tools.msgBox "Entrer 1 point en 3D"
   return -- termine le traitement
end
```

Cette séquence vérifie que la vue courante est bien du type « 3D » et que l'utilisateur a bien introduit 1 point de construction pour positionner l'échelle.

Dans le cas contraire, le programme affiche un message puis termine le traitement.

```
-- (2) Dimensions fixées

local P  = v:getPoint(1) -- position échelle
local de = 27.0 -- distance entre axes échelons
local em = 3.5  -- épaisseur des montants
local pm = 7.0  -- profondeur des montants
local re = 1.75 -- rayon des échelons
local w  = 80.0 -- largeur de l'échelle
```

Dans cet extrait, « **P** » est une variable de type « **Vec3** » (Un vecteur) qui sera utilisé pour positionner l'échelle.

Les variables suivantes sont ici utilisées comme des constantes. Elles fixent les dimensions de l'échelle (sauf sa hauteur).

```
-- (3) Entrer la hauteur désirée

local h, ok = tools.inputBox("Hauteur ?", 0)
if not ok then return; end -- Opération annulée
```

Où « **tools** » est une table *Lua* prédéfinie de *ForthCAD*. Elle contient divers outils (De nouveau...consulter L'API).

```
-- (4) Vérifier hauteur suffisante

if h < 3*de then
   tools.msgBox "Hauteur échelle trop petite"
   return
end
```

Nous abandonnons le traitement si l'échelle est trop petite.

```
-- (5) Construire les montants

local montant = {} -- table de montants
local echelon = {} -- table d'échelons

montant[1] = Atom("BOX3D", P, Vec3(em, h, pm))
montant[2] = Atom("BOX3D",
                  Vec3(P.x+w-em, P.y, P.z),
                  Vec3(em, h, pm))
```

« **montant** » est une table destinée à contenir les 2 montants. Pareillement, les échelons sont stockés dans la table « **échelon** ».

[6] Pour les classes, voir http://www.lua.org/pil/16.html

La classe « **Atom** », prédéfinie dans *ForthCAD,* permet de créer des parallélépipèdes (« **BOX3D** »), pour représenter les deux montants de l'échelle.

```
-- (6) Construire les échelons

local N = h//de -- nombre entier d'échelons.
de = h/(N+1)    -- recalibrer espace échelons
P.x = P.x+em; P.z = P.z+pm/2

local E = Vec3(P.x+w-2*em, P.y, P.z)
for i=1, N do
   P.y = P.y+de
   E.y = P.y
   echelon[i]=Atom("CYLINDER3D", P, E, re)
end
```

Les échelons sont représentés par des cylindres de rayon « **re** » et d'espacement approximatif « **de** ».

La variable « **N** » est initialisée égale au nombre d'échelons, résultat de la division entière de la hauteur de l'échelle par l'intervalle entre les échelons.

Ensuite, « **de** » est recalculé de façon à obtenir une répartition convenable des échelons. (Cette méthode simpliste est discutable...).

Puis les coordonnées du vecteur « **P** » sont modifiées de manière à localiser la partie gauche du premier échelon en bas.

Le vecteur « **local E = ...** » défini la position de la partie droite du premier échelon.

La boucle « **for i=1, N do** » construit les échelons, de bas en haut. Ces échelons sont stockés dans la table « **echelon** ».

```
-- (7) Grouper { {montants} + {échelons} }

local group = Atom("GROUP",
                   { Atom("GROUP", montant),
                     Atom("GROUP", echelon) })
```

Les montants sont groupés. Les échelons sont groupés.

Ces deux groupements sont à leur tour groupés, pour obtenir l'objet final.

Il reste à placer ce groupement dans la vue courante :

```
-- (8) Insérer l'objet dans la vue courante

v:addAtom(group)
```

La méthode « **addAtom** » de la classe « **View** » représentée par l'objet « **v** » ajoute l'objet dans la vue courante.

Remarques

L'échelle de cet exemple n'est pas un objet « intelligent », au sens où ses dimensions ne sont plus sous contrôle après la création.

Ainsi, si la hauteur de l'échelle est modifiée par l'utilisateur, la section des échelons deviendra elliptique.

La programmation d'objets paramétriques « intelligents » sera traitée dans un prochain document.

Classes et objets

ForthCAD défini une quinzaine de classes d'objets qui facilitent la programmation.

Le nom des classes débute toujours par une majuscule (**Vec3**, **Mat3**, **Atom**, **Db**, ...), ce qui permet de les distinguer des tables, comme « **tools** » ou « **geo** », qui encapsulent divers outils.

Vecteur 3D

La classe « **Vec3** » permet de créer un vecteur comme suit :

```
local v = Vec3()
```

« **Vec3** » est en fait une fonction, qui fabrique un objet de type « vecteur 3D ».

Nous pouvons initialiser un vecteur de différentes façons :

```
local v = Vec3(1,2,3)       -- (1)
local w = Vec3(v)           -- (2)
local a = v                 -- (3)
local u = Vec3(10, 20)      -- (4)
local x, y, z = v:unpack() -- (5)
```

Dans cet exemple,

(**1**) Le vecteur « **v** » est construit et initialisé avec les coordonnées **x=1**, **y=2** et **z=3**.

(**2**) Le vecteur « **w** » est ici une copie du vecteur « **v** ».

(**3**) Le vecteur « **a** » est une **référence** au vecteur « **v** ». Il représente le même vecteur que « **v** ». En conséquence, modifier « **v** » modifiera automatiquement « **a** », et inversement !

(**4**) Le vecteur « **u** » est initialisé avec les coordonnées **x=10**, **y=20** et implicitement **z=0**.

(**5**) Les coordonnées du vecteur « **v** » sont récupérées dans les variables **x**, **y**, **z** via « **unpack** ».

Il est possible[7] et fort apprécié d'utiliser des opérateurs sur les vecteurs :

```
local vs = v + w
print(vs)               -- 2,4,6
print(v ^ u)            -- -60, 30, 0
print(v:dot(w))         -- 14
print(v .. w)           -- 14
print ( Vec3(7,8,9)*2 ) -- 14, 16, 18, 0x0
```

L'opérateur « **+** » retourne un nouveau vecteur, égal à la somme de deux vecteurs.

L'opérateur « **^** » effectue le produit vectoriel (« cross product »). Le résultat est un nouveau vecteur.

L'opérateur « **..** » effectue le produit scalaire (« dot product »). Il retourne un nombre. « **dot** » effectue la même opération.

Multiplier ou diviser un vecteur par un scalaire (opérateur « ***** » et « **/** ») renvoie un nouveau vecteur mis à l'échelle.

Il est possible de transformer un vecteur en le multipliant par une matrice « **Mat3** » (Voir § suivant). L'opérateur « ***** » est utilisé dans ce cas.

Remarquez que dans *ForthCAD*, un vecteur 3D possède une quatrième valeur. Il s'agit d'un nombre entier qui permet d'associer un attribut au vecteur.

Matrice 3D

La classe « **Mat3** » crée une matrice de taille 4x4 qui permet de transformer à la fois les vecteurs et les objets graphiques 3D et 2D.

```
local T = Mat3()                          -- (1)
T = T:rotate("y", math.pi/4)              -- (2)
local box = Atom("BOX3D", Vec3(50,55,57)) -- (3)
box:transfo(T)                            -- (4)
```

(**1**) Crée d'une matrice unité 4x4.

(**2**) Multiplie la matrice par une rotation.

[7] http://forthcad.com/fcad_lua_api.html

Il est possible de combiner les séquences (1) et (2) comme suit :

```
local T = Mat3():rotate("y", math.pi/4)
```

(**3**) Création d'un parallélépipède.

(**4**) Rotation du parallélépipède.

Classe « Atom »

La classe « **Atom** » permet de créer des primitives graphiques.

```
local box = Atom("BOX3D", Vec3(50,55,57))       --(1)
local vOrg, vExt = Vec3(0,0), Vec3(0,100)       --(2)
local cyl = Atom("CYLINDER3D", vOrg, vExt, 25.0)--(3)
```

(**1**) Crée un parallélépipède de dimensions données.

(**2**) Défini la position de la base et du sommet du cylindre.

(**3**) Crée un cylindre de rayon **25.0**.

N.B. : Pour faciliter l'écriture, dans la plupart des fonctions qui utilisent des vecteurs en paramètres, ces vecteurs peuvent être remplacés par des tables *Lua* à 2 ou 3 valeurs : **{x,y[,z]}**[8].

```
local cyl = Atom("CYLINDER3D", {0,0}, {0,100}, 25.0)
```

Cet exemple défini le même cylindre que précédemment (où z est implicitement égal à zéro).

```
local cycle = {
  {0,0},
  {80,0,0, 0x1000}, -- début d'arc de cercle
  {100,20},         -- fin de l'arc
  {100,50},
  {0,50}
}
local polyg = Atom("POLYG3D", cycle)
View():addAtom(polyg)
```

Cet exemple utilise les attributs de points de ForthCAD pour définir un polygone comportant un arc de cercle.

Les attributs de points possibles sont :

Point	**Valeur hexa**	**Touche clavier**
Shift	0x8000	Shift
Ctrl	0x4000	Ctrl
Bézier	0x2000	F2
Arc	0x1000	F4

Ces attributs peuvent être combinés de la même façon que pour l'entrée de points de constructions au clavier ou à la souris, dans l'interface de *ForthCAD*.

Les 16 premiers bits de l'attribut sont inutilisés. Ils peuvent servir à numéroter les points dans un but de repérage.

La primitive "**MESH3D**" permet de créer des objets plus complexes avec un minimum de sommets :

```
local vertice =
{
 {0, 0, 0},
 {50,0, 0},
 {25,50,0},
 {0, 0, 30},
 {50,0, 30},
 {25,50,30}
}
local id =
{
 { 1,2,3 },
 { 4,5,6 },
 { 5,2,3,6 },
 { 1,4,6,3 },
 { 4,5,2,1 }
}
View():addAtom( Atom("MESH3D", vertice, id) )
```

Dans cet exemple, l'objet créé est directement affiché dans la vue courante de *ForthCAD*.

Classe « View »

La classe « **View** » permet de gérer une fenêtre de ForthCAD.

```
local v = View()
```

La variable « **v** » contient un objet qui représente la vue courante.

```
local i = 1 -- starting index = 1
while View(i) do
  print(View(i))
  i = i + 1
end
print("View count = "..tostring(i - 1))
```

Cette boucle affiche dans la console, des informations sur les vues ouvertes dans *ForthCAD*.

```
local box = Atom("BOX3D", Vec3(50,55,57)) -- (1)
if View():getDirMatrixState() then        -- (2)
   local T = View():getDirMatrix()        -- (3)
   box:transfo(T)
end
View():addAtom(box)                       -- (4)
```

Cet exemple illustre comment orienter un objet selon un système d'axes (référentiel) défini par l'utilisateur.

(**1**) La primitive « **box** » est créée.

(**2**) Si un référentiel est utilisé...

(**3**) ...Alors récupère la matrice et transforme l'objet.

(**4**) Enfin, ajoute l'objet créé dans la vue courante.

```
local vlist = View():getAtomList()   -- (1)
for atom in vlist:list() do          -- (2)
   print(atom)
end
```

(**1**) « **getAtomList** » retourne une référence vers un objet « groupement », qui contient la scène complète de la vue courante.

(**2**) L'utilisation de l'itérateur « **list**[9] » facilite l'écriture de cette boucle qui affiche tous les objets de la scène dans la console.

———

[8] Dans le manuel, dans l'écriture « **{x, y[, z]}** », les crochets indiquent les valeurs optionnelles. La barre verticale « **|** » indique une alternative.

[9] Utilisés uniquement avec les groupements (Atom"GROUP").

2. Groupement paramétrique

Un groupement paramétrique est un groupement spécial, auquel est associé :

• Un programme *Lua*, qui supervise les caractéristiques, formes et dimensions des composants.

• Une matrice 4x4, qui cumule l'historique des transformations géométriques appliquées à l'objet.

• Eventuellement, des variables.

Quelle que soit la position ou l'orientation de l'objet, le programme associé « voit » toujours cet objet comme contenu dans un parallélépipède rectangle droit, sans rotation ni déformation.

Canevas de programme

Comme pour les autres primitives graphiques, la création de cet objet est réalisée via la classe « **Atom** » :

```
local obj=Atom("LGROUP3D", fileName,typeName,matrix)
```

• « **fileName** » est le nom[10] du fichier qui contient l'interface *Lua* de l'objet.

• « **typeName** » est le nom de la table qui contient l'interface de l'objet. Plusieurs types d'objets peuvent être définis dans un même fichier.

• « **matrix** » est la matrice de transformation initiale de l'objet.

L'objet est initialement vide.

On construit l'objet en ajoutant ses composants dans la liste interne, accessible à l'aide de la fonction « getAtomList ».

Interface

L'interface *Lua* de l'objet permet de répondre à trois évènements, selon le schéma suivant :

```
-- myfile.lua

interface_table =
{
    OnTransfo = function( atom ) --
        ...
    end,
    CanModify = function( atom ) -- boolean
        ...
        return boolResult
    end,
    OnModify = function( atom )  -- boolean
        ...
        return boolResult
    end
}

return { mytype = interface_table }
```

1. OnTransfo (atom)

Cette fonction est exécutée lors des transformations qui modifient les dimensions de l'objet.

En général, cette fonction reconstruit les composants.

2. CanModify (atom) -- boolean

Tenant compte du contexte, généralement basé sur le nombre de points de constructions, cette fonction permet d'accepter ou non la modification de l'objet[11].

3. OnModify (atom) -- boolean

Cette fonction est exécutée en réponse à l'option « **Graphique / Modifier** » du menu.

En général, une boîte de dialogue sera présentée pour modifier des variables.

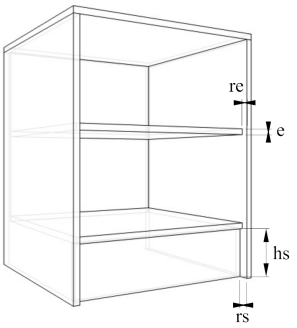
Retourner la valeur booléenne « *true* » pour rafraichir l'objet à l'écran, « *false* » si la modification a été annulée.

Exemple simple

La création d'un corps de meuble simplifié permet d'exposer les concepts de bases sans perte de généralité.

On ignore ici le sens des veines des panneaux, la présence de chants de finitions et les tolérances d'ajustements nécessaires pour le placement des étagères.

Les panneaux seront définis à l'aide de simples parallélépipèdes.



e : épaisseur des panneaux.
hs : hauteur du socle.
rs : retrait avant du socle.
re : retrait avant/arrière étagères.
ef : épaisseur panneau du fond.
Largeur : **60** cm
Hauteur : **75** cm
Profondeur = **55** cm

Afin d'éviter l'utilisation des boîtes de dialogues[12], les dimensions hors-tout de l'objet sont fixées dans le code.

Toutes les dimensions sont ici en [cm][13].

Test du programme (page suivante)

• Sauver le fichier dans « fabric\sample_corpus.lua ».

• Dans la console, taper la commande suivante :

```
require("fabric/sample_corpus").create()
```

Modifier les dimensions hors-tout de l'objet ne modifiera pas la largeur ni la position des panneaux, ce qui était le but recherché.

Cliquant sur l'équerre « Graph / Modifier » permet de modifier l'épaisseur des panneaux.

[10] Utiliser le chemin d'accès relatif au répertoire de travail de *ForthCAD,* sans extension.

[11] Activation ou non de l'équerre de la barre d'outils et de l'option « Graphique / Modifier » du menu.

[12] La création de boîtes de dialogues est traitée dans un chapitre à part.

[13] L'utilisation d'un coefficient d'échelle permet d'utiliser un programme écrit en [cm] dans un environnement qui travaille en [mm], et inversement.

Code Lua

```lua
-- sample_corpus.lua
local function buildCorpus( atom )
   local sx, sy, sz = atom:getSize():unpack()
   local e = atom:get("e", 1.8)
   local hs = 15
   local re = 2
   local rs = 3
   local ef = 0.4
   local group = atom:getAtomList()

  -- Set/Replace component
   local function Set(ref, vOrg, vSize)
      local elem = Atom( "BOX3D", vOrg, vSize )
      elem:setName(ref)
      local old = group:listFindName(ref)
      if old and group:listReplace(old, elem) then
         elem:setColor(old:getColor())
         old:destroy()
      else
         group:listAddHead(elem)
      end
   end
   -- joues gauche/droite
   local s = {e, sy-e, sz}
   Set("JG",    {0, 0, 0},  s)
   Set("JR",    {sx-e, 0, 0}, s)
   -- top
   Set("TOP",   {0, sy-e, 0}, {sx, e, sz})
   -- etagères
   s = {sx-2*e, e, sz-2*re}
   Set("ETAG0", {e, hs, re}, s)
   Set("ETAG1", {e, (sy+hs-e)/2, re}, s)
   -- socle
   Set("SOCLE", {e, 0, sz-rs-e}, {sx-2*e, hs, e})
   -- fond
   Set("FOND",  {e, hs, re-ef},
                {sx-2*e, sy-hs-e, ef})
end

local interface_corpus =
{
   OnTransfo = function( atom )
      buildCorpus( atom )
   end,
   CanModify = function( atom ) -- bool
      return true -- enable modification
   end,
   OnModify = function( atom ) -- bool
      local e = atom:get("e", 1.8)
      local ok
      e,ok = tools.inputBox("Epaisseur panneaux", e )
      if ok then
         atom:set("e", e)
         buildCorpus( atom )
      end
      return ok
   end
}

local function create()
   if View():sn()~=3 then return end
   local atom = Atom("LGROUP3D",
          "fabric\\sample_corpus", "corpus",
          Mat3():scale(60,75,55))
   buildCorpus(atom)
   View():addAtom(atom)
end

return { corpus=interface_corpus, create=create }
```

Description du programme

• En fin de programme, la fonction « **create** » fabrique un objet de classe « **Atom** ». Le paramètre « **LGROUP3D** » précise la sous-classe de l'objet.

```lua
local function create()
   if View():sn()~=3 then return end
   local atom = Atom("LGROUP3D",
          "fabric\\sample_corpus", "corpus",
          Mat3():scale(60,75,55))
   ...
```

La fonction « **buildCorpus** » ajoute les composants de l'objet.

Finalement, « **View():addAtom** » ajoute le corpus à la vue courante :

```lua
   buildCorpus(atom)
   View():addAtom(atom)
end
```

• Au début du programme, la fonction « **buildCorpus** » reçoit en paramètre l'objet créé, d'abord vide de composants.

```lua
local function buildCorpus( atom )
   local sx, sy, sz = atom:getSize():unpack()
   local e = atom:get("e", 1.8)
   local hs = 15
   local re = 2
   local rs = 3
   local ef = 0.4
   local group = atom:getAtomList()
```

En préambule, on initialise les différentes constantes décrites au début de ce paragraphe :

• Les variable « **sx, sy, sz** » correspondent à la taille hors-tout du corpus. Ces valeurs sont retournées par la méthode « **getSize** ».

• L'épaisseur des panneaux, « **e** », est la seule variable qui soit modifiable dans cet exemple. Cette variable est stockée dans l'objet à l'aide de la méthode « **set** » et récupérée à l'aide de la méthode « **get** », qui renvoie ici la valeur par défaut, 1.8 cm.

• La variable « **group** » est l'objet « groupement » intégré dans l'objet. Elle est utilisée par la fonction « **Set** » pour ajouter ou remplacer les composants dans le groupement.

```lua
  -- Set/Replace component
   local function Set(ref, vOrg, vSize)
      local elem = Atom( "BOX3D", vOrg, vSize )
      elem:setName(ref)
      local old = group:listFindName(ref)
      if old and group:listReplace(old, elem) then
         elem:setColor(old:getColor())
         old:destroy()
      else
         group:listAddHead(elem)
      end
   end
```

• « **elem** » est un parallélépipède dont la taille correspond à celle d'un panneau. Chaque panneau est repérable par sa référence.

• « **old** » est l'ancienne version de ce parallélépipède. Si ce panneau existe, on récupère sa couleur avant de le remplacer.

---

3. Primitive Lua

Une primitive graphique *Lua* est une entité graphique unique, à laquelle sont associés :

• Un programme *Lua*, qui permet la création et la modification de la géométrie de l'objet, au moyen de facettes triangulaires ou rectangulaires.

• Une matrice 4x4, qui cumule l'historique des transformations géométriques appliquées à l'objet.

• Eventuellement, des variables internes à l'objet.

• Eventuellement, un stock de matériaux en complément du matériel de colorisation de base.

Comme pour le « groupement paramétrique » du chapitre précédent, le programme associé à la primitive « voit » toujours la géométrie comme si elle était contenue dans un parallélépipède rectangle droit, sans rotation ni déformation.

Canevas de programme

La création de cet objet est réalisée via la classe « **Atom** » :

```
local obj = Atom("LOBJ3D", fileName,typeName,matrix)
```

La seule différence avec le « groupement paramétrique » est ici l'utilisation de la sous-classe « LOBJ3D ».

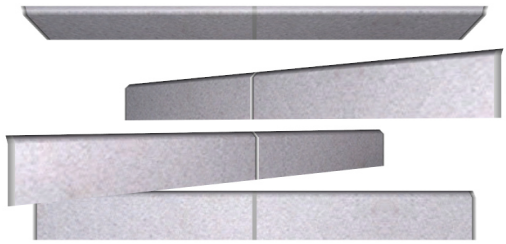
Interface

L'interfaçage est identique à celui utilisé pour le « groupement paramétrique » du chapitre précédent :

```
-- myfile.lua
interface_table =
{
    OnTransfo = function( atom ) --
        ...
    end,
    CanModify = function( atom ) -- boolean
        ...
        return boolResult
    end,
    OnModify = function( atom )  -- boolean
        ...
        return boolResult
    end
}

return { mytype = interface_table }
```

Exemple simple

On se propose de créer une plinthe formée d'une série d'éléments de carrelage de tailles ~ 30 x 8 x 1 [cm].



Les joints de 0.6 [cm] de largeur sont réalisés de manière à rendre un effet de relief, bien visible sur l'image.

Dans ForthCAD, on positionnera cette plinthe à l'aide de 2 points de constructions P1 et P2.

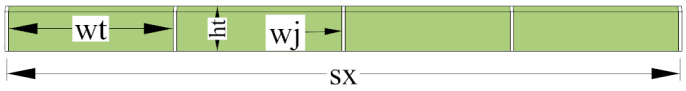
Un peu de calcul...

• Calcul de nombre de carreaux et de joints nécessaires.

Soient **sx, ht, et** la largeur, hauteur (8.0) et épaisseur (1.0) de la plinthe.

Soit **N** : Le nombre de carreaux de la plinthe.

Soit **wt, wj** : La largeur (30) d'un carreau et largeur (0.6) d'un joint.



Avec un joint à chaque extrémité, on a

$$sx = N * (wt + wj) + wj$$

C.-à-d.

$$N = (sx - wj) / (wt + wj)$$

Comme **N** est généralement un nombre décimal, et pour ne pas compliquer cet exemple, on utilise la simplification suivante[14] :

$$N = (sx - wj) // (wt + wj) + 1$$

Où **N** est un nombre entier. La largeur d'un carreau devient :

$$wt = (sx - wj) / N - wj$$

• Calcul de la longueur et orientation de la plinthe.

Puisque les plinthes sont toujours horizontales, on peut effectuer les calculs suivants, basés sur deux points (vecteurs **p1**, **p2**) de constructions entrés dans *ForthCAD* :

L'origine étant en **p1**, soit le vecteur :

$$dp = p2 - p1$$

L'ordonnée de **p2** doit être identique à celle de **p1**. D'où :

$$dp.y = 0$$

La longueur « **sx** » de la plinthe est :

$$sx = |dp|$$

La rotation « **alfa** » de la plinthe est :

$$alfa = \arctan(-dp.z / dp.x)$$ [15]

La matrice de transformation **T** d'une plinthe unité placée à l'origine des axes est calculée comme suit :

$$T = Scale(sx, ht, et)$$

$$T = T * Rotate('y', alfa)$$

$$T = T * Translate(p1)$$

Le code *Lua* correspondant serait :

```
T = Mat3():scale(sx,ht,et):rotate('y',alfa):translate(p1)
```

En pratique, on multiplie « **ht** » et « **et** » par un facteur « **ku** » égal à **10** si on travaille en unités [mm], et **1** en unités [cm].

[14] L'opérateur « // » effectue une division entière.

[15] On utilise « math.atan ». Consulter l'API de *Lua* 5.7.

Code Lua

```
-- dimensions
local ku = tools.getUnits()==2 and 10.0 or 1.0
local wt = ku*30.0    -- largeur carreaux
local ht = ku*8.0     -- hauteur carreaux
local et = ku*1.0     -- épaisseur carreaux
local wj = ku*0.6     -- largeur joints

local function buildPlinth( a )
   a:scale(Vec3(1)/a:getSize())-- scale correction
   a:clear()------------------- initialize object
   local sx, sy, sz = a:getSize():unpack()
   local N = (sx-wj)//(wt+wj)+1
   wt = (sx-wj)/N-wj -- new tile width

   local function tile(x, w, nx)
      local function vpair(x, y, z, ny, nz)
         a:normal(nx, ny, nz)
         a:pd(x, y, z)
         a:normal(-nx, ny, nz)
         a:pd(x+w, y, z)
      end
      a:useColor(nx==0 and 0 or "white")
      a:Begin('Q')
         vpair(x, 0, sz, 0,1)
         vpair(x, sy-sz, sz, 0,1)
         vpair(x, sy, sz/2, 1,1)
         vpair(x, sy, 0, 1,0)
      a:End()
   end

   tile(0, wj, 1)
   local x = wj
   for i=1,N do
      tile(x, wt, 0)
      tile(x+wt, wj, 1)
      x = x + wt + wj
   end
   -- points attraction
   a:attractor(a:vertex(0, 0))
   a:attractor(a:vertex(sx,0))
end

local interface =
{
   -- CanModify = function (a) return false; end,
   -- OnModify  = function (a) return false; end,
   OnTransfo = function (a) buildPlinth(a); end
}

local function create()
   local v=View()
   if v==nil or v:sn()~=3 or v:getPointCount()~=2
   then return; end

   local p1 = v:getPoint(1)
   local dp = v:getPoint(2)-p1
   dp.y = 0
   local len = dp:length()
   if len<3*wj then return; end        -- too small
   local alfa = math.atan(-dp.z, dp.x)

   local T = Mat3():scale(len, ht, et) -- dimensions
   T = T:rotate('y', alfa):translate(p1)
   local a=Atom('LOBJ3D','fabric/plinth','plinth',T)

   buildPlinth(a)
   View():addAtom(a)
end

return    { plinth = interface, create = create }
```

Description du programme

• On ne permet pas la modification, la table « **interface** » est donc réduite à l'évènement « **OnTransfo** ».

• Le programme débute par la déclaration de constantes multipliées par un facteur « **ku** » qui tient compte des unités utilisées :

```
local ku = tools.getUnits()==2 and 10.0 or 1.0
```

La fonction « **tools.getUnits()** » renvoie « **2** » si les unités sont en [mm].

• Fonction « **create** » : Après vérification du contexte, cette fonction calcule la matrice de transformation initiale puis crée l'objet (Voir détail calculs page 8).

• Fonction « **buildPlinth** » : Cette fonction construit ou reconstruit la plinthe.

Elle débute par une mise à l'échelle[16] :

```
   a:scale( Vec3(1)/a:getSize() )-- scale correction
```

ForthCAD permet, en cours de construction, de transformer la géométrie à l'aide d'une pile de matrices. La matrice au sommet de la pile multiplie implicitement les points, les normales, et les coordonnées de textures.

Cette pile est initialisée avec une matrice unité[17].

Dans la définition de « **create** », on a vu que la matrice « **T** » de transformation de l'objet est multipliée par des coefficients d'échelles qui correspondent aux dimensions de la plinthe :

```
   local T = Mat3():scale(len, ht, et) -- dimensions
```

Or, la fonction « **buildPlinth** » construit la géométrie de l'objet en utilisant aussi les dimensions réelles de la plinthe. Sans précautions, ces dernières seraient donc élevées au carré !

Il est donc nécessaire de débuter la construction en multipliant la matrice au sommet de la pile par des coefficients d'échelles valant l'inverse des dimensions de l'objet.

• Fonction « **tile (x, w, nx)** »

Cette fonction crée un carreau ou un joint en position « **x** » et de largeur « **w** ».

La valeur de « **nx** » sert à la fois à modifier la normale[18] des joints dans la fonction « **vpair** » et à coloriser l'élément comme un joint (si **nx=1**), ou comme un carreau (si **nx=0**). Nous détaillons cette fonction plus loin.

```
      a:useColor(nx==0 and 0 or "white")
```

La fonction « **useColor** » permet l'utilisation de matériaux prédéfinis (« white », « black », « glass »...) ou du matériel « **0** », défini par l'utilisateur.

La séquence « **nx==0 and 0 or "white"** » est un idiome[19] propre à *Lua* pour effectuer une sélection.

La séquence finale limite les points d'attractions aux deux extrémités de la plinthe.

```
   a:attractor(a:vertex(0, 0))
   a:attractor(a:vertex(sx,0))
```

[16] « getSize » renvoie les coefficients d'échelles dans un vecteur.

[17] Voir API Lua dans ForthCAD : « pushMatrix », « popMatrix », « multMatrix », etc.

[18] La « shading normal » qui sert à l'éclairement.

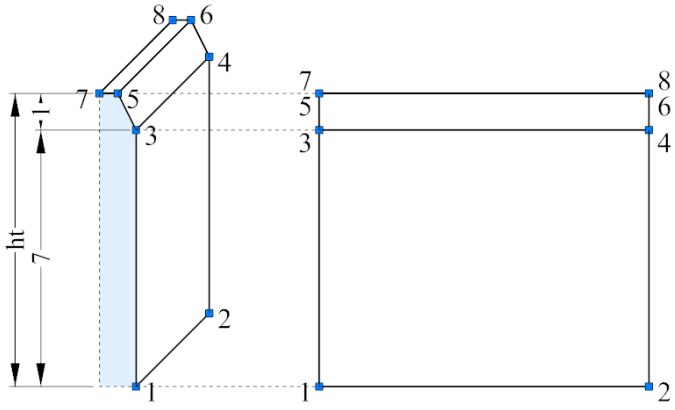
[19] Voir http://www.lua.org/pil/3.3.html

Description détaillée de la fonction « tile ».

Cette fonction utilise des notions non vues précédemment. Elle est recopiée ici :

```
   local function tile(x, w, nx)
      local function vpair(x, y, z, ny, nz)
         a:normal(nx, ny, nz)
         a:pd(x, y, z)
         a:normal(-nx, ny, nz)
         a:pd(x+w, y, z)
      end
      a:useColor(nx==0 and 0 or "white")
      a:Begin('Q')
         vpair(x, 0, sz, 0,1)
         vpair(x, sy-sz, sz, 0,1)
         vpair(x, sy, sz/2, 1,1)
         vpair(x, sy, 0, 1,0)
      a:End()
   end
```

Pour construire un carreau ou un joint, on utilise ici seulement trois quadrilatères :



- quadrilatère 1 : {1,2,3,4}
- quadrilatère 2 : {3,4,5,6}
- quadrilatère 3 : {5,6,7,8}

Pour alléger cet élément décoratif, les quadrilatères arrière et en bas sont omis. De même les 2 polygones latéraux sont omis.

Pour définir les quadrilatères, on utilise la paire de fonctions « **Begin('Q')** » ... « **End()** ». Les points sont entrés à l'aide de la fonction « **pd(...)** »[20].

On aurait pu définir le carreau de la façon suivante :

```
-- gauche           droite --
a:Begin'Q'
   a:pd(0,0,1);     a:pd(wt,0,1)     -- 1-2
   a:pd(0,7,1);     a:pd(wt,7,1)     -- 3-4 (quad 1)
   a:pd(0,8,0.5);   a:pd(wt,8,0.5)   -- 5-6 (quad 2)
   a:pd(0,8,0);     a:pd(wt,8,0)     -- 7-8 'quad 3)
a :End()
```

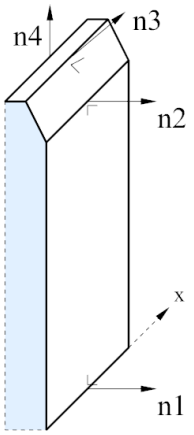
On voit que **N** quadrilatères sont ainsi définis à l'aide de **2N+2** points.

Ce dernier exemple est correct pour un rendu filaire, mais pas pour un rendu avec éclairage, où il est nécessaire de définir la normale aux surfaces pour moduler la réaction de chaque face à la lumière.

Bien que le calcul automatique de la normale soit possible, ce confort ne permettrait pas d'en altérer la direction, dans le but de moduler les réactions à la lumière.

Dans le programme, on a incliné la normale pour donner un aspect arrondi au dessus de chaque carreau et de chaque joint.

Cet effet est obtenu en utilisant les orientations indiquées sur la figure suivante.



Le pseudo-code correspondant est

```
Begin('Q')
  normal(0,0,1)
  pd(1); pd(2)
  normal(0,0,1)
  pd(3); pd(4)
  normal(0,1,1)
  pd(5); pd(6)
  normal(0,1,0)
  pd(7); pd(8)
End()
```

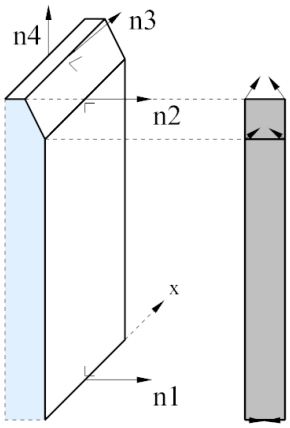
Le résultat obtenu est le suivant :

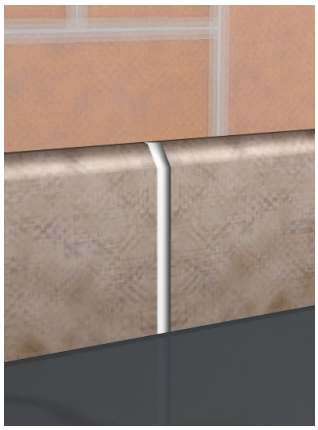
L'effet d'arrondi est bien visible avec une source située à droite et vers le haut.

Le rendu peut encore être amélioré en faussant les normales latéralement, lorsque c'est le joint qui est dessiné.

Pour les joints, une inclinaison des normales vers l'intérieur est illustrée sur la figure suivante.

Le résultat est montré sur l'image de droite.





Un effet de profondeur est ainsi suggéré pour le joint.

Sans cette technique, il faudrait un grand nombre de facettes pour obtenir un rendu quasi identique.

---

[20] « **pd** » = « **p**en **d**own ». Le paramètre est un vecteur.

4. Page 2D

Des outils 2D *Lua* permettent d'automatiser la mise en page, de créer des légendes, de numéroter ou de nommer les objets.

Plutôt que d'énumérer ces fonctions, qui sont documentées dans l'API, on se propose de décrire un exemple simple, duquel on pourra s'inspirer.

Création d'un cartouche

Cette séquence dessine un cartouche d'inscriptions avec des textes fixés.

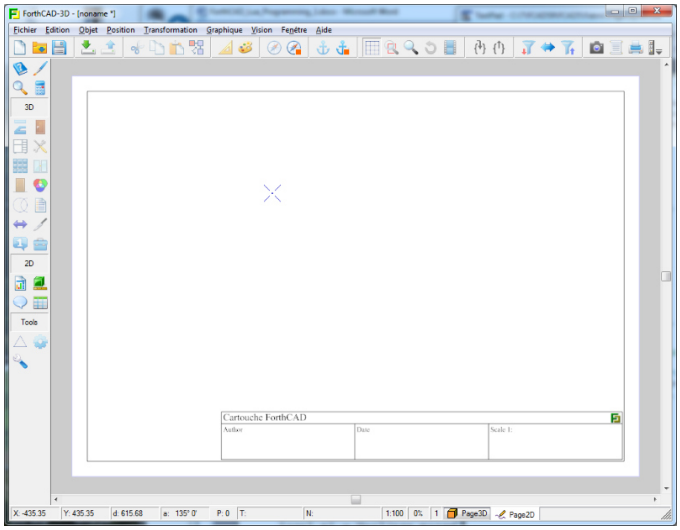
```
-- (1) Check context

local v = View()
if not v or v:sn()~=2 then
   tools.msgBox("Usage in 2D context")
   return
end

-- (2) Text edition (to define)

local function DialogText()
   return {
      Title = "Cartouche ForthCAD",
      "Author : ", "Date : ", "Scale : 1/"
   }
end
local txt = DialogText()
if not txt then return; end -- cancel

-- (3) Reads page informations

local pge = v:getPageInfo()
local kE = 1./pge.scale
local p1 = Vec3(pge.marge_left, pge.marge_bottom)
local p2 = Vec3(pge.width - pge.marge_right,
                pge.height - pge.marge_top)

-- (4) Computes positions and sizes

if p2.x-p1.x > 230 then p1.x = p2.x-212; end
local x, y, h, w = p1.x+1, p1.y+1, 25, p2.x-p1.x-2
local UP = 0x8000 -- point-shift mask

-- (5) Frame (thickness 0.2)

local frame =
{
   {x, y}, {x, y+h}, {x+w, y+h}, {x+w, y}, {x, y},
   -- title
   {x, y+3*h/4, 0, UP}, {x+w, y+3*h/4,0},
   -- divisions 1|2|3
   {x+1*w/3, y, 0, UP}, {x+1*w/3, y+3*h/4},
   {x+2*w/3, y, 0, UP}, {x+2*w/3, y+3*h/4}
}

local atom = Atom("LINE2D", frame)
atom:transfo( Mat3():scale(kE) )
atom:setStyle2D{ lineWidth = 2 }
v:addAtom( atom )

-- (6) Text function. Page metrics in [mm]
local function TextAt(x, y, size, txt)
   local a = Atom("TEXT2D", txt, size)
   a:transfo(Mat3():translate(kE*Vec3(x,y)))
   v:addAtom(a)
end
-- (7) Creates title block texts

TextAt(x+1, y+h, 14, txt.Title)
for i=1,3 do
   TextAt(x+w/3*(i-1)+1, y+3*h/4-1, 10, txt[i])
end

-- (8) Loads/sets logo image (.\LOGO.JPG)

local strFile = tools.getStartDir("LOGO.JPG")
local vPos = kE*Vec3(x+w-h/5-1, y+3*h/4+0.5)
atom = Atom("IMG2D", strFile, vPos, kE*h/5)
v:addAtom( atom )
```

Nous verrons plus loin comment réaliser une boîte de dialogue pour l'édition des textes.

Description du programme

Pour tester ce programme, copier le texte dans le fichier « fabric/tools.lua ». Dans l'éditeur *SciTE*, presser **F5**.

Le cartouche devrait se présenter comme suit :



• Après vérification du contexte (**1**) et l'édition des textes (**2**) dans une boîte de dialogue (à définir, voir § suivant), le programme lit (**3**) le format et l'échelle utilisée pour la page courante :

```
-- (3) Reads page informations

local pge = v:getPageInfo()
```

Les dimensions retournées par la fonction « **getPageInfo** » sont en [mm].

On calcule les positions « **p1** » et « **p2** » des coins inférieur-gauche et supérieur-droit de la zone imprimable.

• A partir de ces valeurs, on calcule en (**4**) :

x, y : coordonnées gauche et inférieure du cartouche.

h, w : hauteur et largeur du cartouche.

• En (**5**), la table « **frame** » contient les coordonnées de tracé de la grille du cartouche.

Le tracé est ensuite effectué sur base d'un objet « **Atom** » de la sous-classe « **LINE2D** ».

```
local atom = Atom("LINE2D", frame)
```

Pour convertir les unités « papier » en unités réelles, on applique la transformation d'échelle « **kE** » à l'objet.

```
atom:transfo( Mat3():scale(kE) )
```

L'épaisseur des lignes est ensuite fixée à 0.2 [mm].

```
atom:setStyle2D{ lineWidth = 2 }
```

L'objet est finalement ajouté dans la vue courante.

```
v:addAtom( atom )
```

• En (**6**), on défini une fonction qui permet de positionner le texte en coordonnées papier [mm]. La taille du texte « **size** » est définie en points typographiques.

```
local function TextAt(x, y, size, txt)
   local a = Atom("TEXT2D", txt, size)
   a:transfo(Mat3():translate(kE*Vec3(x,y)))
   v:addAtom(a)
end
```

• Cette fonction est ensuite utilisée en (**7**) pour placer le texte dans les quatre zones du cartouche :

```
TextAt(x+1, y+h, 14, txt.Title)
for i=1,3 do
   TextAt(x+w/3*(i-1)+1, y+3*h/4-1, 10, txt[i])
end
```

• Finalement (**8**), une image « LOGO.JPG », à placer dans le répertoire de *ForthCAD*, est chargée et positionnée à droite de la zone de titre du cartouche.

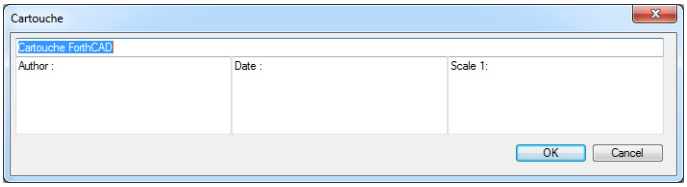
```
-- (8) Loads/sets logo image (.\LOGO.JPG)

local strFile = tools.getStartDir("LOGO.JPG")
local vPos = kE*Vec3(x + w - h/5 - 1, y + 3*h/4 +0.5)
atom = Atom("IMG2D", strFile, vPos, kE*h/5)
v:addAtom( atom )
```

Pour tenir dans la zone de titre, le logo doit ici être de forme carrée.

Notez que l'absence du fichier « LOGO.JPG » ne produira pas d'erreur. En effet, si l'objet est égal à « **nil** », la fonction « **addAtom** » abandonne simplement le traitement.

Création de la boîte de dialogue

Dans l'exemple précédent, on a laissé en suspend la définition de la boîte de dialogue destinée à éditer les textes du cartouche.



Voici une définition possible, qui remplace celle du programme précédent :

```
local function DialogText ()
   local txt =
   {
      Title = "Cartouche ForthCAD",
      "Author", "Date", "Scale 1:"
   }
   do
      local ini = Ini"FCAD-5.INI"
      ini:section"cartouche"
      txt.Title = ini:get("Title", txt.Title)
      txt[1] = ini:get("Text1", txt[1])
      txt[2] = ini:get("Text2", txt[2])
      txt[3] = ini:get("Text3", txt[3])
   end
   local ret = Dialog
   {
      title = "Cartouche",
      message = function(dlg, id, value)
         if id=='cancel' then
            dlg:exit(0)
         elseif id=='ok' then
            local ini = Ini"FCAD-5.INI"
            ini:section"cartouche"
            txt.Title = dlg:get'0'
            txt[1] = dlg:get'1'
            txt[2] = dlg:get'2'
            txt[3] = dlg:get'3'
            ini:set("Title", txt.Title)
            ini:set("Text1", txt[1])
            ini:set("Text2", txt[2])
            ini:set("Text3", txt[3])
            dlg:exit(1)
         end
      end,
      -- Dialogbox template --
      { id='0',   ctrl='edit',   text=txt.Title,
        style='',   x=3,   y=3, cx=450, cy=12 },
      { id='1',   ctrl='edit',   text=txt[1],
        style='m',  x=3,    y=15,cx=150, cy=50 },
      { id='2',   ctrl='edit',   text=txt[2],
        style='m',  x=153,    y=15,cx=150, cy=50 },
      { id='3',   ctrl='edit',   text=txt[3],
        style='m',  x=303,    y=15,cx=150, cy=50 },
      { id='ok',       ctrl='button', text='OK',
        x=350,   y=70,   cx=50,   cy=12 },
      { id='cancel', ctrl='button', text='Cancel',
        x=403,   y=70,   cx=50,   cy=12 }
   }
   return ret==1 and txt or nil
end
```

Description succincte du code Lua

• Les textes par défauts sont lus dans la section « **[cartouche]** » du fichier d'initialisation « **FCAD-5.INI** », situé dans le répertoire de *ForthCAD*.

```
   do
      local ini = Ini"FCAD-5.INI"
      ...
   end
```

L'utilisation de « **do ... end** » permet d'isoler la variable « **ini** » et de limiter sa durée de vie, afin de fermer le fichier dés qu'il n'est plus nécessaire[21].

La fonction « **Ini:get(var, default)** » retourne une valeur de même type que la valeur par défaut renseignée en second paramètre.

• La construction de la boîte de dialogue est réalisée à l'aide d'un canevas (Une table *Lua*) passé en paramètre à la fonction « **Dialog** » (Voir API).

Cette table est structurée comme suit :

- title : Titre de la boîte.

- **message** : Fonction « boucle de message » utilisée par *ForthCAD*.

- dialog template : Une série de tables qui renseignent les propriétés des champs de contrôle.

[21] Voir api Lua - http://www.lua.org/pil/4.2.html