ATELIER

Créer une interface graphique en pur code

* Mise en pratique Lua / Love2D *

Type d'atelier :
Centre Technologique

Technos :
Lua / Love2D

Lien de l'atelier :

Sommaire

Introduction et objectifs

Prérequis : Avoir lu et assimilé l'introduction à l'atelier

Ceci est la 2ème partie de l'atelier pour la mise en pratique Lua/Love2D.

Mise en pratique (Lua)

Note aux développeurs C# / C++

Je vous propose de considérer cette mise en pratique comme une approche théorique. Le Lua étant quasiment aussi lisible que du pseudo-code.

Fonctions utiles

Afficher le curseur de la souris : love.mouse.setVisible
Récupérer les coordonnées de la souris : love.mouse.getPosition
Autres fonctions liées à la souris : https://love2d.org/wiki/love.mouse

Attention : si vous utilisez un système de type viewport pour gérer le full screen et les changements de résolution, pensez à adapter le calcul de la position de la souris.

Design

Je propose qu'on place toutes les fonctions GUI dans une table nommée GCGUI, qui sera renvoyée par un fichier Lua séparé. Cela permet d'avoir une approche POO, au moins dans la forme.

Créez donc un fichier GCGUI.lua dans votre projet et ajoutez ce code pour créer un groupe :

```
local GCGUI = {}

return GCGUI
```

Gérer les groupes

Voici le code pour créer un groupe, avec ses fonctions de base (à insérer dans GCGUI.lua, avant le return) :

```
function GCGUI.newGroup()
  local myGroup = {}
  myGroup.elements = {}

  function myGroup:addElement(pElement)
    table.insert(self.elements, pElement)
```

```
  end

  function myGroup:setVisible(pVisible)
    for n,v in pairs(myGroup.elements) do
      v:setVisible(pVisible)
    end
  end

  function myGroup:draw()
    love.graphics.push()
    for n,v in pairs(myGroup.elements) do
      v:draw()
    end
    love.graphics.pop()
  end

  return myGroup
end
```

Notez l'utilisation du push / pop dans le draw. Cela permet de sauvegarder le contexte graphique à l'entrée et de le restaurer à la fin. Intérêt ? Vous pouvez faire plein de setColor ou des transformations pendant le draw de votre GUI, tout sera remis comme avant à la fin.

Pour utiliser la GUI, il suffira de faire ceci dans main.lua (une seule fois au début, en dehors de toute fonction) :

```
local myGUI = require("GCGUI")
```

puis quand vous voudrez créer un groupe :

```
myGroup = myGUI.newGroup()
```

Observez les fonctions setVisible et draw. Nous prévoyons donc que nos éléments exposent tous une fonction "setVisible" et "draw".

EXERCICE : En Lua, il est facile de vérifier l'existence d'une fonction. Améliorez le code de setVisible et draw pour qu'il vérifie si l'élément expose bien la fonction prévue avant de l'appeler.

Notre fonction d'usinage d'élément

Comme je vous l'ai enseigné, nous allons utiliser une fonction "usine" pour créer notre table de base, qui pourra ensuite être enrichie pour gérer les éléments de GUI.

Elle proposera de base une coordonnée, un état "Visible" et une fonction setVisible.

Dans GCGUI.lua :

```
local function newElement(pX, pY)
  local myElement = {}
  myElement.X = pX
  myElement.Y = pY
  function myElement:draw()
    print("newElement / draw / Not implemented")
  end
  function myElement:setVisible(pVisible)
    self.Visible = pVisible
  end
  return myElement
end
```

Premier élément : panel

Un panel est juste un panneau avec une position et une taille, qui va recevoir une image pour le représenter.

Ajoutez une fonction à la table GCGUI :

- qui usine un nouvel élément
- qui l'enrichit d'une largeur (W) et une hauteur (H), et d'une image (nil au départ)
- qui expose une fonction setImage recevant une image Love
- implémentez la fonction draw

Note : regardez comment je sépare draw / drawPanel. Pourquoi ? Cela permettra de décliner le Panel en conservant la possibilité d'accéder à drawPanel.

```
function GCGUI.newPanel(pX, pY, pW, pH)
  local myPanel = newElement(pX, pY)
  myPanel.W = pW
  myPanel.H = pH
  myPanel.Image = nil

  function myPanel:setImage(pImage)
    self.Image = pImage
    self.W = pImage:getWidth()
    self.H = pImage:getHeight()
  end

  function myPanel:drawPanel()
    love.graphics.setColor(255,255,255)
    if self.Image == nil then
      love.graphics.rectangle("line", self.X, self.Y, self.W, self.H)
    else
```

```
      love.graphics.draw(self.Image, self.X, self.Y)
    end
  end

  function myPanel:draw()
    if self.Visible == false then return end
    self:drawPanel()
  end

  return myPanel
end
```

A noter aussi que si une image est fournie, je prend sa taille comme référence. Aucune raison d'avoir un panel graphique d'une taille différente de son image.

Pour tester, nous allons retourner dans main.lua :

Déclarez d'abord une globale pour votre groupe, en haut en dehors de toute fonction :

```
local groupTest
```

Dans load :

```
  panelTest1 = myGUI.newPanel(10, 350, 300, 200)

  groupTest = myGUI.newGroup()
  groupTest:addElement(panelTest1)
```

Dans draw :

```
  groupTest:draw()
```

Vous devriez voir votre panel sous forme de cadre blanc.

Pour avoir une image, utilisez le panel "panel1.png" du projet de démo, de cette manière :

```
  panelTest1 = myGUI.newPanel(10, 350)
  panelTest1:setImage(love.graphics.newImage("panel1.png"))
```

Note : je ne fournis pas la taille car elle sera calculée par setImage !

Jouez avec les valeur, créez d'autres panels, pour bien comprendre l'architecture de votre système. N'oubliez pas d'ajouter vos nouveaux panels au groupe.

Pour la suite, je vous laisse tester les nouveaux éléments sur le même modèle. Référez-vous au projet de démo si vous voulez voir un code finalisé.

Texte

Nous allons créer un élément Text capable de recevoir une police de caractère (Font) et d'aligner horizontalement et/ou verticalement le texte dans une zone donnée.

On est donc proche d'un Panel. Et devinez quoi ? Il suffit d'usiner un Panel au lieu d'un Element afin de bénéficier des propriétés du panel, en l'occurrence sa taille.

Les spécificités demandant un peu de réflexion sont :

- La police de caractère
- L'alignement du texte

Pour la police nous allons simplement la créer en dehors de la GUI et la passer en paramètre. Ainsi chaque élément Text pourra avoir une police différente si besoin.

Pour l'alignement, j'ajoute 2 paramètres recevant une chaîne de caractère, un pour l'horizontal, l'autre pour le vertical. On pourra ainsi gérer des variantes telles que "center", "right", "left"... Je vais déjà gérer "left" et "center", sachant que "left" est par défaut si on ne précise pas l'alignement.

Voici le code que je propose :

```
function GCGUI.newText(pX, pY, pW, pH, pText, pFont, pHAlign, pVAlign)
  local myText = GCGUI.newPanel(pX, pY, pW, pH)
  myText.Text = pText
  myText.Font = pFont
  myText.TextW = pFont:getWidth(pText)
  myText.TextH = pFont:getHeight(pText)
  myText.HAlign = pHAlign
  myText.VAlign = pVAlign

  function myText:drawText()
    love.graphics.setColor(255,255,255)
    love.graphics.setFont(self.Font)
    local x = self.X
    local y = self.Y
    if self.HAlign == "center" then
      x = x + ((self.W - self.TextW) / 2)
    end
    if self.VAlign == "center" then
      y = y + ((self.H - self.TextH) / 2)
    end
    love.graphics.print(self.Text, x, y)
  end
```

```
  function myText:draw()
    if self.Visible == false then return end
    self:drawText()
  end

  return myText
end
```

Exercice

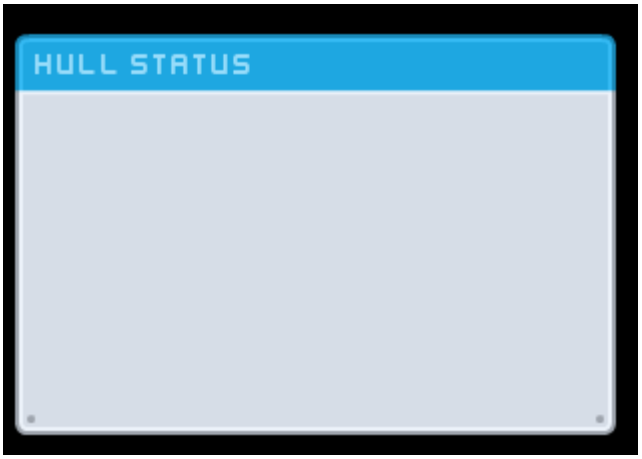
1) Ajoutez la possibilité de spécifier la couleur du texte.
2) Rendez ce paramètre optionnel
3) Ajoutez un titre à votre 1er panel, et centrez-le verticalement au bon endroit (dans la zone de titre)

Indice 1 : une variable complexe permet de stocker plusieurs valeurs dans une seule variable.

Indice 2 : lorsqu'un ou plusieurs paramètres (en fin de liste) ne sont pas transmis à une fonction, leur valeur est alors "nil" dans le corps de la fonction.

Indice 3 : Utilisez un logiciel de dessin (Photoshop, Gimp, Paint.net) pour mesurer la taille de la zone de titre de l'image panel1.png.

Le résultat :



Bouton

Voilà un élément plus complexe !

Un bon bouton de GUI doit gérer plusieurs choses :

- Une image différente quand il est normal et quand il est enfoncé
- Un événement quand on le clique (et idéalement quand on le relâche)
- Un événement quand on le survole (enter / exit) (optionnel)

- Une image différente quand il est survolé (optionnel)

Notre système a besoin de quelques améliorations :
- Gérer un update en plus du draw
- Intégrer l'update dans le groupe

Déjà, on ajoute un update par défaut dans newElement :

```
  function myElement:update(dt)
    print("newElement / update / Not implemented")
  end
```

Puis on intègre une boucle d'update dans le groupe :

```
  function myGroup:update(dt)
    for n,v in pairs(myGroup.elements) do
      v:update(dt)
    end
  end
```

Et pensez à ajouter un appel à l'update du groupe dans main.lua :

```
function love.update(dt)
  groupTest:update(dt)
end
```

Voilà on est prêts !

Voici déjà le code pour un bouton basique, sans image et sans comportement. Il s'agit juste d'un panel avec un texte par dessus ! Remarquez comme la modularité et la complémentarité de notre code rend la création d'un nouvel élément aisée.

```
function GCGUI.newButton(pX, pY, pW, pH, pText, pFont, pColor)
  local myButton = GCGUI.newPanel(pX, pY, pW, pH)
  myButton.Text = pText
  myButton.Font = pFont
  myButton.Label = GCGUI.newText(pX, pY, pW, pH, pText, pFont,
                                 "center", "center", pColor)
  myButton.isHover = false
  myButton.isPressed = false

  function myButton:draw()
    if self.isPressed then
```

```
      self:drawPanel()
      love.graphics.setColor(255,255,255,50)
      love.graphics.rectangle("fill",
                              self.X, self.Y, self.W, self.H)
    elseif self.isHover then
      self:drawPanel()
      love.graphics.setColor(255,255,255)
      love.graphics.rectangle("line",
                              self.X+2, self.Y+2, self.W-4, self.H-4)
    else
      self:drawPanel()
    end
    self.Label:draw()
  end

  return myButton
end
```

Ajoutons maintenant les comportements "hover" (survolé) et "pressed" (cliqué).

Voici les propriétés dont nous avons besoin, ajoutons-les :

```
  myButton.isHover = false
  myButton.isPressed = false
  myButton.oldButtonState = false
```

Juste 2 booléens pour les 2 états, et un booléen tout aussi important pour gérer le changement d'état du bouton de la souris.

Il s'agit de pouvoir détecter, dans l'update du bouton, si le bouton était enfoncé précédemment ou pas ! Sans cette astuce, l'utilisateur pourrait cliquer en dehors du bouton puis glisser vers le bouton en maintenant le bouton enfoncé, et boum, le bouton se clique alors que ce n'est pas le cas !

Ajoutons une fonction update :

```
  function myButton:update(dt)
    local mx,my = love.mouse.getPosition()
    if mx > self.X and mx < self.X + self.W and
       my > self.Y and my < self.Y + self.H then
      if self.isHover == false then
        self.isHover = true
      end
    else
```

```
    if self.isHover == true then
      self.isHover = false
    end
  end

  self.oldButtonState = love.mouse.isDown(1)
end
```

Ne vous laissez pas impressionné, tout est simple ici. C'est juste que tout arrive d'un coup !

On regarde juste si le curseur de la souris se situe dans les bornes du bouton (c'est un peu comme un test de collision mais avec un seul point), et on change isHover à true (vrai) si c'est le cas. Si le curseur est hors de la zone du bouton, et qu'il était en isHover avant, on repasse l'état à false (faux).

Note : Pourquoi je ne passe isHover à faux que si il était à vrai avant ? Par soucis d'optimisation ? Non… C'est parce que je projette d'avoir un événement déclenché si la souris quitte le bouton, et je ne veux le déclencher qu'un seule fois !

Allez ! On ajoute le comportement pour le clic (pressed), juste après le "end" du traitement du hover, et avant la sauvegarde du oldButtonState :

```
  if self.isHover and love.mouse.isDown(1) and
      self.isPressed == false and
      self.oldButtonState == false then
    self.isPressed = true
  else
    if self.isPressed == true and love.mouse.isDown(1) == false then
      self.isPressed = false
    end
  end
```

Regardez comment je vérifie sur le bouton était bien non enfoncé. En résumé, pour déclencher le clic du bouton (et changer son état isPressed), on doit avoir ces conditions remplies :

- L'élément de GUI doit être actuellement survolé (hovr)
- Le bouton de la souris doit être enfoncé (isDown)
- Le bouton de la souris ne devait pas être enfoncé lors du précédent update (oldButtonState)

Ensuite, si on constate que le bouton était pressé mais que le bouton de la souris n'est pas enfoncé, on change l'état isPressed du bouton à faux (et bientôt on déclenchera un événement).

Voici un affichage bricolé, puisque pour l'instant nous n'avons pas d'images pour représenter notre bouton. On dessine donc des boites :

```
function myButton:draw()
  if self.isPressed then
    self:drawPanel()
    love.graphics.setColor(255,255,255,50)
    love.graphics.rectangle("fill", self.X, self.Y, self.W, self.H)
  elseif self.isHover then
    self:drawPanel()
    love.graphics.setColor(255,255,255)
    love.graphics.rectangle("line",
                            self.X+2, self.Y+2, self.W-4, self.H-4)
  else
    self:drawPanel()
  end
  self.Label:draw()
end
```

On ajoute 3 images ?

- Une pour l'état normal (default)
- Une pour l'état survolé (hover)
- Une pour l'état cliqué (pressed)

Voici les propriétés à ajouter à l'usinage de l'objet :

```
myButton.imgDefault = nil
myButton.imgHover = nil
myButton.imgPressed = nil
```

Et la fonction pour les initialiser :

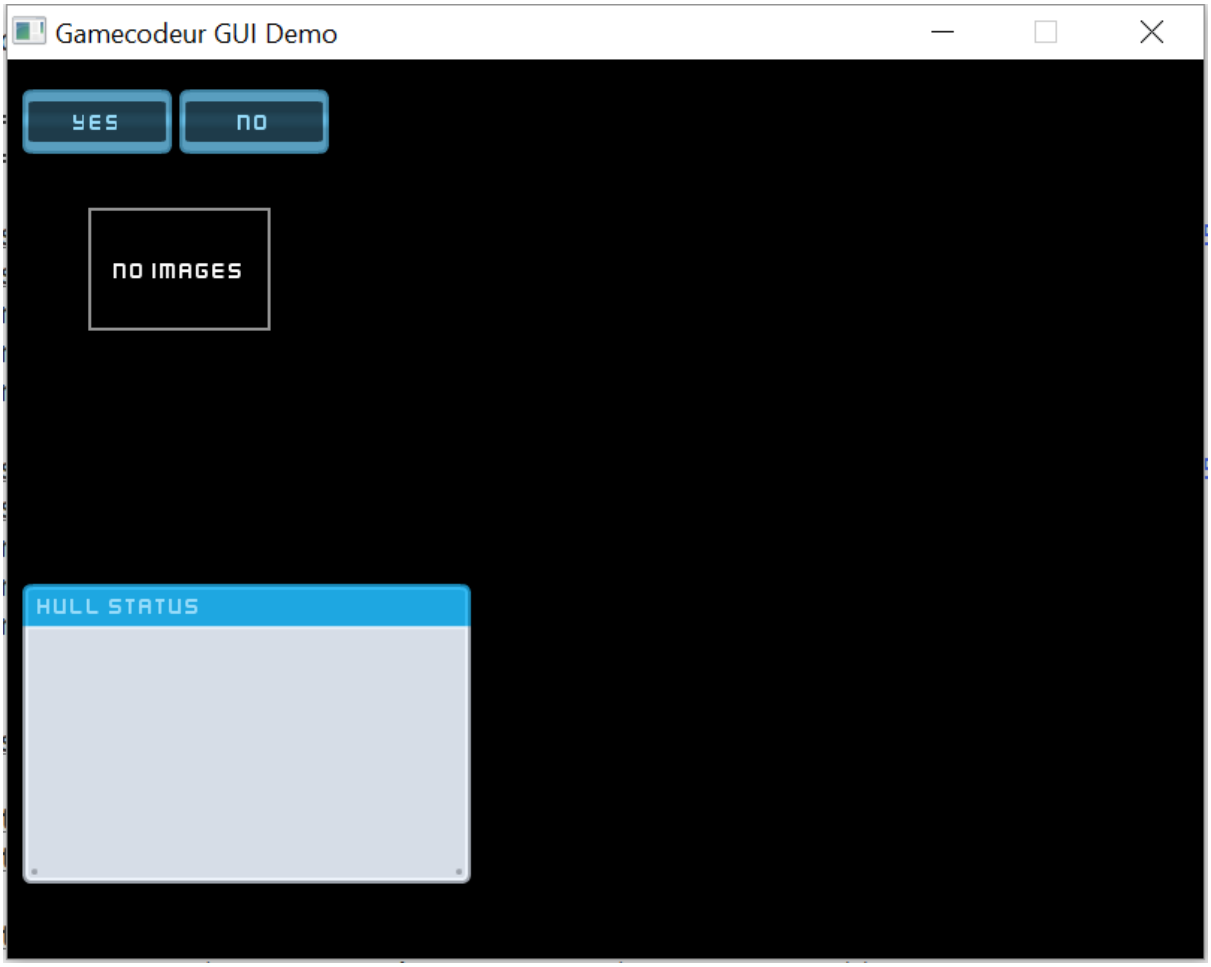
```
function myButton:setImages(pImageDefault, pImageHover, pImagePressed)
  self.imgDefault = pImageDefault
  self.imgHover = pImageHover
  self.imgPressed = pImagePressed
  self.W = pImageDefault:getWidth()
  self.H = pImageDefault:getHeight()
end
```

Note : Si vous ne voulez pas gérer plusieurs images, passez la même aux 3 paramètres.

Nous allons maintenant améliorer le draw pour qu'il soit capable d'afficher, ou pas, les images (selon si elles ont été fournies ou non) :

```
function myButton:draw()
  love.graphics.setColor(255,255,255)
  if self.isPressed then
    if self.imgPressed == nil then
      self:drawPanel()
      love.graphics.setColor(255,255,255,50)
      love.graphics.rectangle("fill", self.X, self.Y, self.W, self.H)
    else
      love.graphics.draw(self.imgPressed, self.X, self.Y)
    end
  elseif self.isHover then
    if self.imgHover == nil then
      self:drawPanel()
      love.graphics.setColor(255,255,255)
      love.graphics.rectangle("line",
                              self.X+2, self.Y+2, self.W-4, self.H-4)
    else
      love.graphics.draw(self.imgHover, self.X, self.Y)
    end
  else
    if self.imgDefault == nil then
      self:drawPanel()
    else
      love.graphics.draw(self.imgDefault, self.X, self.Y)
    end
  end
  self.Label:draw()
end
```

En gros, si on a une image on l'affiche, sinon on continue à dessiner nos boîtes pourries.



Fabuleux non ?

Gérer les événements

Un bouton sans événement ce n'est pas un bouton.

On va déjà commencer par déplacer la gestion du isHover dans le panel, puisque rappelez-vous, nous avions prévu que tous les éléments issus de panneaux puissent le gérer.

```
myButton.isHover = false
```

est déplacé au niveau du panel :

```
myButton.isHover = false
```

Ensuite, on déplace l'algo qui teste l'état isHover dans l'update spécifique du panel :

```
function myPanel:updatePanel(dt)
```

```
    local mx,my = love.mouse.getPosition()
    if mx > self.X and mx < self.X + self.W and my > self.Y and my <
self.Y + self.H then
      if self.isHover == false then
        self.isHover = true
      end
    else
      if self.isHover == true then
        self.isHover = false
      end
    end
  end
```

Et on le remplace par ceci au tout début de l'update du bouton :

```
    self:updatePanel(dt)
```

En dernier lieu, on implémente l'update par défaut :

```
  function myPanel:update(dt)
    self:updatePanel()
  end
```

Pour déclencher un événement, la technique la plus simple est de fournir à l'élément de GUI une référence de fonction.

En Lua c'est trivial puisqu'on peut passer une fonction en paramètre !

On ajoute donc, au niveau du panel une liste pour stocker une table associant un type d'événement à une fonction :

```
  myPanel.lstEvents = {}
```

Ensuite, une fonction pour paramétrer l'événement, elle se contente d'ajouter un élément à notre liste, en associant le type (utilisé comme index) à la fonction :

```
  function myPanel:setEvent(pEventType, pFunction)
    self.lstEvents[pEventType] = pFunction
  end
```

Dans main.lua, on va créer une fonction et la passer à la GUI :

```
function onPanelHover(pState)
  print("Panel is hover :"..pState)
end
```

Puis :

```
  panelTest1 = myGUI.newPanel(10, 350, 300, 200)
  panelTest1:setImage(love.graphics.newImage("panel1.png"))
  panelTest1:setEvent("hover", onPanelHover)
```

Notre table contiendra donc :

"hover"	Référence de la fonction onPanelHover

Dernière étape, brancher les autres événements sur le bouton :

```
  function myButton:update(dt)
    self:updatePanel(dt)
    if self.isHover and love.mouse.isDown(1) and
        self.isPressed == false and
        self.oldButtonState == false then
      self.isPressed = true
      if self.lstEvents["pressed"] ~= nil then
        self.lstEvents["pressed"]("begin")
      end
    else
      if self.isPressed == true and love.mouse.isDown(1) == false then
        self.isPressed = false
        if self.lstEvents["pressed"] ~= nil then
          self.lstEvents["pressed"]("end")
        end
      end
    end
    self.oldButtonState = love.mouse.isDown(1)
  end
```

Exercice :

- Branchez des fonctions aux événements "pressed" de vos boutons !

Cases à cocher

Une case à cocher est quasiment un bouton… C'est juste qu'il reste enfoncé tant qu'on ne re-clique pas dessus. Elle n'a pas de texte non plus (libre à vous d'en afficher un à côté).

Ce ne va donc pas nous prendre beaucoup de temps. On va dupliquer le bouton, enlever l'image sur le hover, le texte et gérer la permanence de l'état pressed.

Exercice :

Créez l'élément Checkbox sans regarder ma proposition, puis comparez.

Voici ma proposition :

```
function GCGUI.newCheckbox(pX, pY, pW, pH)
  local myCheckbox = GCGUI.newPanel(pX, pY, pW, pH)
  myCheckbox.Text = pText
  myCheckbox.imgDefault = nil
  myCheckbox.imgPressed = nil
  myCheckbox.isPressed = false
  myCheckbox.oldButtonState = false

  function myCheckbox:setImages(pImageDefault, pImagePressed)
    self.imgDefault = pImageDefault
    self.imgPressed = pImagePressed
    self.W = pImageDefault:getWidth()
    self.H = pImageDefault:getHeight()
  end

  function myCheckbox:setState(pbState)
    self.isPressed = pbState
  end

  function myCheckbox:update(dt)
    self:updatePanel(dt)
    if self.isHover and love.mouse.isDown(1) and
        self.isPressed == false and
        self.oldButtonState == false then
      self.isPressed = true
      if self.lstEvents["pressed"] ~= nil then
        self.lstEvents["pressed"]("on")
      end
    elseif self.isHover and love.mouse.isDown(1) and
        self.isPressed == true and
        self.oldButtonState == false then
      self.isPressed = false
      if self.lstEvents["pressed"] ~= nil then
        self.lstEvents["pressed"]("off")
      end
```

```
    end
    self.oldButtonState = love.mouse.isDown(1)
  end

  function myCheckbox:draw()
    love.graphics.setColor(255,255,255)
    if self.isPressed then
      if self.imgPressed == nil then
        self:drawPanel()
        love.graphics.setColor(255,255,255,50)
        love.graphics.rectangle("fill", self.X, self.Y, self.W, self.H)
      else
        love.graphics.draw(self.imgPressed, self.X, self.Y)
      end
    else
      if self.imgDefault == nil then
        self:drawPanel()
      else
        love.graphics.draw(self.imgDefault, self.X, self.Y)
      end
    end
  end

  return myCheckbox
end
```

La principale différence est dans la façon de gérer le clic. Observez le code il ne devrait pas vous poser de problème.

Barres de progression

Pour dessiner une barre de progression, nous allons utiliser un simple rectangle, qui se remplira de 0 à 100%. Notre barre sera par contre capable de calculer sa proportion toute seule !

Exemple : votre bouclier possède 300 points d'énergie maximum. Si il ne lui reste plus que 50 points, la proportion est de 16,6 %. La barre sera donc remplie à 16,6 % mais nous n'aurons pas à faire ce calcul.

Un peu de mathématiques

La formule pour calculer une proportion est hyper simple :

Proportion = Niveau / Total

Donc pour l'exemple donné plus haut :

Proportion = 50 / 300
Proportion = 0,16666 (j'arrondis à 5 chiffres après la virgule pour plus de lisibilité)

Si notre barre fait 170 pixels de long, sa taille sera de :

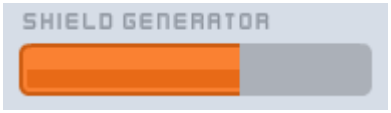
170 x 0,16666 = 28,33 pixels !

Visuellement, le plus simple est d'avoir un cadre, et de le remplir.

Certains graphistes vont vous proposer des barres comme celle-ci :

Dans ce cas, il y a un défaut. Comme vous le voyez la barre orange est arrondie à droite et à gauche ! L'arrondi mesure 6 pixels. Donc avec les 2 côtés on obtient une taille minimale de 12 pixels. Comment représenter une barre de 4 pixels ?!

La solution est plutôt de cropper (tronquer) la barre :



On peut ainsi avoir la précision nécessaire.

Je vous propose de coder les 2 :
- Si on ne fournit pas d'images à l'objet, il dessinera une boite
- Si on lui fournit les images, il se chargera de cropper l'image

Dans le cas d'un version "image" de la barre, je propose qu'on fournisse :
- le fond
- la barre complète

Les 2 ayant la même taille, on a juste à les superposer.

On code ?

Je vais déjà ajouter une couleur au Panel afin de l'utiliser proprement comme cadre pour les Progress Bar sans images.

On ajoute un paramètre au constructeur :

```
function GCGUI.newPanel(pX, pY, pW, pH, pColor)
```

puis on stocke la valeur :

```
...
myPanel.ColorOut = pColorOut
...
```

On l'utilise ensuite dans le drawPanel :

```
    if self.ColorOut ~= nil then
      love.graphics.setColor(self.ColorOut[1],
                             self.ColorOut[2],
                             self.ColorOut[3])
    else
      love.graphics.setColor(255,255,255)
    end
```

Voilà, on peut maintenant avoir une barre de progression basique :

```
function GCGUI.newProgressBar(pX, pY, pW, pH, pMax, pColorOut, pColorIn)
  local myProgressBar = GCGUI.newPanel(pX, pY, pW, pH)
  myProgressBar.ColorOut = pColorOut
  myProgressBar.ColorIn = pColorIn
  myProgressBar.Max = pMax
  myProgressBar.Value = pMax

  function myProgressBar:setValue(pValue)
    if pValue >= 0 and pValue <= self.Max then
      self.Value = pValue
    else
      print("myProgressBar:setValue error - out of range")
    end
  end

  function myProgressBar:draw()
    self:drawPanel()
    local barSize = (self.W - 2) * (self.Value / self.Max)
    if self.ColorOut ~= nil then
      love.graphics.setColor(self.ColorIn[1],
```

```
                              self.ColorIn[2],
                              self.ColorIn[3])
    else
      love.graphics.setColor(255,255,255)
    end
    love.graphics.rectangle("fill",
                            self.X + 1, self.Y + 1, barSize, self.H - 2)
  end

  return myProgressBar
end
```

Pour l'ajouter à notre scène :

```
progressTest = myGUI.newProgressBar(panelTest1.X + 35,
                                    panelTest1.Y + 68, 220, 26, 100,
                                    {50,50,50}, {250, 129, 50})
...
groupTest:addElement(progressTest)
```

On peut y ajouter un titre :

```
title1 = myGUI.newText(panelTest1.X + 35, panelTest1.Y + 45, 0, 0,
                       "Shield Generator", mainFont, "", "",
                       {157, 164, 174})
...
groupTest:addElement(title1)
```

Et faire baisser la valeur progressivement sans love.update :

```
  if progressTest.Value > 0 then
    progressTest:setValue(progressTest.Value - 0.01)
  end
```

Impressioné(e) ? !

Ajoutons la gestion des images :

```
  ...
  myProgressBar.imgBack = nil
  myProgressBar.imgBar = nil

  function myCheckbox:setImages(pImageBack, pImageBar)
```

```
    self.imgBack = pImageBack
    self.imgBar = pImageBar
    self.W = pImageBack:getWidth()
    self.H = pImageBack:getHeight()
  end
```

Pour l'affichage nous allons afficher le fond tel quel, par contre nous devons cropper (tronquer) la barre !

Nous allons pour cela utiliser un Quad qui permet de spécifier une zone d'une image, et ensuite de l'afficher. Voici le nouveau début de la fonction draw de la progress bar :

```
  function myProgressBar:draw()
    love.graphics.setColor(255,255,255)
    local barSize = (self.W - 2) * (self.Value / self.Max)
    if self.imgBack ~= nil and self.imgBar ~= nil then
      love.graphics.draw(self.imgBack, self.X, self. Y)
      local barQuad = love.graphics.newQuad(0, 0,
                                    barSize, self.H, self.W, self.H)
      love.graphics.draw(self.imgBar, barQuad, self.X, self. Y)
    else
      ... ancien code ...
```

Nous créons donc d'abord un Quad de la taille raccourcie :

```
local barQuad = love.graphics.newQuad(0, 0,
                                    barSize, self.H, self.W, self.H)
```

(voir l'aide de la fonction newQuad : https://love2d.org/wiki/love.graphics.newQuad)

Puis nous passons ce Quad en 2ème paramètre de la fonction love.graphics.draw :

```
love.graphics.draw(self.imgBar, barQuad, self.X, self. Y)
```

Cette syntaxe étant une variante permettant l'utilisation d'un Quad pour spécifier la portion de l'image à afficher (voir https://love2d.org/wiki/love.graphics.draw).

C'est tout. Notre progress bar est fonctionnelle et ne représente qu'une 40e de lignes de code !

Exercice

Gérez la possibilité d'avoir une barre verticale en plus de horizontale.

Exercices pour aller plus loin

Gérer une hiérarchie Groupe / Elements

Créez une hiérarchie, et ainsi gérez une position relative au groupe :

- Un groupe possède une position
- L'ajout d'un élément se fait relativement à la position du groupe
  (si l'élément est en x=10, c'est 10 pixels par rapport à la position x du groupe et non à l'écran)
- Le déplacement d'un groupe impacte la position des éléments qu'il contient

Ajoutez du tweening et du blending

- Prérequis conseillé : gérer une hiérarchie groupe / éléments
- Un groupe peut s'afficher avec un effet de tweening
- Un groupe peut s'afficher avec un effet de blending (fade-in)
  (indice : gérez simplement une valeur pour l'alpha, que chaque élément va appliquer, et qui part de 0 pour aller à 100% sur une durée)

Gérer des boutons 9-Slices pour qu'ils soient redimensionnables

Voici le principe :

- On fournit 9 images
- Seuls les 4 coins sont figés
- Les 5 autres parties seront "scalées" afin de respecter la taille du bouton
- Les côtés gauche et droite sont scalés verticalement
- Les côtés haut et bas sont scalés horizontalement
- Le centre est scalé verticalement et horizontalement

En images :