Développement d'un MOTUS avec Python / Kivy

Table des matières

1) La structure générale

Pour la structure générale, j'ai pensé à créer un tableau de tableaux de labels, qui sont placés dans les cellules. De cette manière, je peux modifier le contenu de chaque label en utilisant ses coordonnées en 2 dimensions (qui ne changent pas).
Ces doubles coordonnées me permettent facilement d'implémenter le code du jeu.

a. Kivy ou python ?

Problème : je ne sais pas si je dois écrire le code dans le kv ou dans le py.

Solution :

C'était un questionnement pendant une grande partie du projet.
Maintenant je suis au clair là-dessus :
kivy pour tout l'aspect graphique et statique
Python pour la dynamique (modifier les valeurs, gérer les évènements)

b. Properties et bindings

Problème : comment accéder aux widgets et à leurs évènements ?

Pendant une grande partie de mon projet, j'ai été confronté à des problèmes de ce type.
Comment désactiver le textinput ? Comment appeler l'instance depuis la classe parente ?
Pour les bindings : comment récupérer un paramètre envoyé par un évènement ? Si la méthode (listener) est déclarée dans la classe du Widget ou même la classe parente, cela n'est pas suffisant.

Solution :

La solution a été de tout restructurer.
Car au départ, j'avais une partie des classes (Widgets) dans le kv et une autre partie était ajoutée depuis le python (avec add_widget). Je pensais que de cette manière je pourrais accéder aux classes enfants. C'est vrai, mais c'était compliqué de s'y retrouver.
Et puis j'ai compris le système des ObjectProperty.

C'est simple : la structure statique du projet est écrite entièrement en kv :

```
<GameLayout>:

  my_text_input: my_textInput
  my_grid: my_grid
  my_bt_validate: id_bt_validate
  my_bt_replay: id_bt_replay
  my_bt_settings: id_settings

  MainBoxLayout:
    MainGrid:
      id: my_grid
    InputBox:
      MyTextInput:
        id: my_textInput
      RoundButton:
        id: id_bt_validate
        text: "OK"
        bold: True
        font_size: dp(25)
        size_hint: None, None
        size: dp(70), dp(70)
        back_color: hex(cst.color_bleu_3) if self.state == 'normal' else hex(cst.violet_1)
    BottomLayout:
      cols: 3
      rows: 1
```

Les Widgets à utiliser sont référencés grâce à une id et liés chacun à une ObjectProperty au niveau de la classe super-conteneur (GameLayout ou SettingsLayout). Cette classe a donc, un rôle de contrôleur, en plus de servir à la disposition des Widgets enfants. C'est même cette fonction de contrôleur qui domine largement. C'est elle qui accède à toutes les variables et qui va implémenter les bindings.

Dans le python :

```
class GameLayout(FloatLayout):

    my_text_input = ObjectProperty()
    my_grid = ObjectProperty()
    my_bt_validate = ObjectProperty()
    my_bt_replay = ObjectProperty()
    my_bt_settings = ObjectProperty()
```

Problème : comment accéder aux évènements ?

Lorsque l'ObjectProperty cible un widget enfant du layout, le binding (dans le init) ne pose pas de problème. Mais lorsqu'on veut créer une connexion avec un widget d'une classe enfant, on obtient une erreur, disant que l'objet vaut null.

Solution :

Il apparaît que c'est un problème courant (heureusement, comme cela trouver la solution a été rapide). Le référencement par id n'a pas lieu au moment de l'init, donc les variables ne contiennent rien. C'est pourquoi le binding ne marche pas. Une solution (la solution ?) est de différer les bindings à l'aide de Clock.

```
def __init__(self, **kwargs):
    super(GameLayout, self).__init__(**kwargs)

    self.rows = cst.NB_LINES
    self.cols = 6

    Clock.schedule_once(self._init_later)

def _init_later(self, *args):
    self.init_game()
    self.init_audio()
    self.my_text_input.bind(on_text_validate=self.on_text_validate)
    self.my_bt_replay.bind(on_press=self.on_press_replay)
    self.my_bt_validate.bind(on_press=self.on_bt_validate)
```

Sans argument additionnel (en plus du callback) Clock.schedule_once appelle le callback à la frame suivante :

`schedule_once`(*callback*, *timeout=0*)

Schedule an event in <timeout> seconds. If <timeout> is unspecified or 0, the callback will be called after the next frame is rendered.

Vous remarquerez que je diffère aussi init_game et init_audio. La raison est que init_game appelle my_grid et my_textinput également (et qu'ils valent None). Pour le init_audio, c'était juste pour tout regrouper.

2) Les couleurs

a. Couleur de fond (background)

Problème : comment changer la couleur de la frame principale ? (elle est noire par défaut)

Solution 1 :

Ajout d'un canvas.before au niveau du layout principal.

```
<MainGrid>:
  canvas.before:
    Color:
      rgba: (.31, .86, 1, 1)
    Rectangle:
      pos: self.pos
      size: self.size
```

Solution 2 :

Modifier un attribut de la classe Window :

```
from kivy.core.window import Window

Window.clearcolor = get_color_from_hex(cst.BG_MAIN)
```

b. Couleur au format kivy

Problème : Comment convertir une couleur au format kivy ?

Solution 1 :

Pour une couleur un format hexadécimal :
Dans le kv, on peut utiliser la méthode get_color_from_hex

```
#:import hex kivy.utils.get_color_from_hex
```

```
canvas.before:
    Color:
        rgba: hex('#58AE6F')
```

Autre possibilité :

L'en-tête comporte un import et les constantes :

```
#:import utils kivy.utils
#:set color_orange "#DD7835"
```

```
color: utils.get_color_from_hex(color_orange)
```

Remarque :

utils possède une méthode pour obtenir une couleur random :

```
<ColorCanva>:
    canvas.before:
        Color:
            rgba: utils.get_random_color()
        Rectangle:
            pos: self.pos
            size: self.size
```

Dans Python :

```
with self.canvas.before:
    Color(rgba=utils.rgb_to_kvColor(cst.BG_FRAME_COLOR))
    Rectangle(pos=self.pos, size=self.size)
```

On peut écrire directement le tuple ou écrire une fonction comme je l'ai fait ici, qui prend un code RGB et le met au format kivy.

Voici le code :

```
def rgb_to_kvColor(rgb):
    value = [x / 255 for x in rgb]
    if len(value) == 3:
        value.append(1)
    return value
```

Pour chaque élément de la liste, on divise par 255. Si on n'a que 3 éléments, on ajoute la valeur pour la couche alpha = 1

et les constantes décrivant la couleur :

```
# set rgb colors
BG_CELL_COLOR = '#%02x%02x%02x' % (30, 160, 240)
BG_FRAME_COLOR = '#%02x%02x%02x' % (80, 220, 255)
```

3) L'affichage des labels

a. Afficher un background pour chaque label

Problème : Comment avoir un background pour chaque label ?

Solution :

Au début, j'ai créé une classe de label avec un canva et un rectangle qui s'adapte à la taille du label.

```
class MyLabel(Label):

    def on_size(self, *args):
        self.canvas.before.clear()
        with self.canvas.before:
            Color(.12, .63, .94, 1)
            Rectangle(pos=self.pos, size=self.size)
```

Un problème est vite apparu : au niveau du python, toucher au canva demande plusieurs étapes. Non seulement effacer, puis redessiner, mais il faut aussi penser au redimensionnement (donc modifier le canva dans l'évènement on_size, par exemple).

Dans le kv c'est beaucoup plus simple (donc plus adapté), parce qu'on n'a pas à s'occuper du redimensionnement. La boucle de kivy s'en occupe. Une fois que qu'on définit la canvas, cela se fait automatiquement.

C'est pourquoi j'opte pour cette approche : kivy s'occupe du dessin et python du dynamisme. Dans le kv, les couleurs sont donc des variables, déclarées dans le python comme Listproperty (puisque ce sont des tuples).

kv :

```
<CustomLabel>:
    canvas.before:
        Color:
            rgba: self.bg_color
        Rectangle:
            pos: self.pos
            size: self.size
        Color:
            rgba: self.bg_color_2
        Ellipse:
            pos: self.pos
            size: self.size
```

python :

```
class CustomLabel(Label):
    bg_color = ListProperty()
    bg_color_2 = ListProperty()

    def __init__(self, **kwargs):
        super(CustomLabel, self).__init__(**kwargs)
        self.mode = 0

        self.font_family = "Courrier"
        self.font_size = dp(50)
        self.bg_color = utils.rgb_to_kvColor(cst.BG_CELL_COLOR)
        self.bg_color_2 = cst.TRANSPARENT

    def change_mode(self, mode):
        self.mode = mode
        if mode == 0:
            self.bg_color = utils.rgb_to_kvColor(cst.BG_CELL_COLOR)
            self.bg_color_2 = cst.TRANSPARENT

        elif mode == 1:
            self.bg_color = utils.rgb_to_kvColor(cst.BG_CELL_COLOR)
            self.bg_color_2 = utils.rgb_to_kvColor(cst.BG_CELL_COLOR_ORANGE)

        else:
            self.bg_color = utils.rgb_to_kvColor(cst.BG_CELL_COLOR_RED)
            self.bg_color_2 = cst.TRANSPARENT
```

b. Dessiner un cercle (ou pas)

Pour l'ajout du disque, je dessine une ellipse dans une cellule au format carré.

Pour gérer l'affichage du disque, je dessine le disque mais je fais varier la valeur de l'alpha : il vaut 0 quand je ne veux pas afficher le disque (code 0 ou 2) et il vaut 1 quand je veux l'afficher (code 1). La modification se fait en modifiant bg_color_2 dans le python.

J'ai mis un certain temps avant de penser à cette astuce (j'étais concentré l'ajout de l'ellipse uniquement dans le python).

4) La taille des cellules dans la grid

Problème : avoir des cellules carrées, qui peuvent s'agrandir ou se rétrécir avec la fenêtre

J'en ai bien bavé.

Pour avoir des cellules carrées, soit on fixe la taille des cellules - mais dans ce cas elles ne s'adapteront pas à la taille de la fenêtre - ou on fixe une valeur (la hauteur ou la largeur) par rapport à l'autre.

J'ai cette méthode, mais j'ai rencontré des problèmes.

```
class MyLabel(Label):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.mode = 0
        self.bg_color = cst.BG_CELL_COLOR


        self.size_hint = None, 1
        # widget.size_hint = (width_percent, height_percent)
        self.width = self.height
        self.size_hint_min_y = dp(50)
```

Sachant que ces labels sont dans un gridLayout
Le width est fixé par rapport au height
la hauteur maximale des cellules est fixée à 50

Problèmes :
- Tant que je ne redimensionne pas la fenêtre en hauteur (y), les cellules ne s'affichent pas.
- Avec des cellules carrées, la largeur de la grid est plus large que les carrés.
- Ce n'est pas joli car la couleur du fond déborde.
- La fenêtre ne s'adapte pas aux cellules.
- La couleur et les layout, tout part en vrille

Si on s'occupe juste de faire les cellules carrées, le pbm est que la grid ne se redessine pas, parce qu'on ne lui donne pas les dimensions adaptées.

J'ai aussi essayé d'adapter la taille de la fenêtre principale.

Je fixe une window size min pour pouvoir afficher les lettres :

```
Window.size = ((70 + self.padding[0]) * self.cols + 70, (70 + self.padding[1]) * self.rows + 70)
Window.minimum_width, Window.minimum_height = Window.size
```

Comment détecte une modification de width ou de height pout adapter la taille de la fenêtre, afin de conserve le ratio ?

Il faudrait faire un bind pour détecter le changement de taille de la fenêtre et considérer les cas : x qui augmente, x qui diminue, y qui augmente, y qui diminue

Mais je ne trouve pas de méthode évènement qui permette de récupérer ces paramètres.

Solution :

Fixer une proportion pour la grid.

Dans le kv :

```
<MainGrid>:
  size_hint_y: None
  size_hint_x: 1
  height: self.width * self.rows / self.cols

  padding: [dp(5), dp(5), dp(5), dp(5)]
  spacing: [dp(5), dp(5)]

  canvas.before:
    Color:
      rgba: utils.rgb_to_kvColor(cst.BG_FRAME_COLOR)
    Rectangle:
      pos: self.pos
      size: self.size
```

le height est calculé à partir du width (variable selon la taille de la fenêtre) et du nombre de lignes et de colonnes. Une simple règle de 3 afin de garder le ratio.

5) Le text input

a. Validation et lien avec la grid

Problème : comment afficher le contenu du TextIput dans la grid ?

Solution :

Grace aux exemples du cours, j'ai cela :

```
<InputBox>:
  orientation: "horizontal"
  spacing: "10dp"
  TextInput:
    id: my_textInput
    size_hint: None, 1
    width: "100dp"
    text: "Bonjour"
    multiline: False
    on_text_validate: root.on_text_validate(self)
  Label:
    text: root.text_input
```

Lors de la validation, on appelle **on_text_validate(self)** du root, c'est-à-dire de la classe conteneur, InputBox. Dans la classe InputBox, on déclare une **StringProperty**. Elle est appelée dans le label avec :

```
text: root.text_input
```

Sauf que j'ai besoin, lors du **on_text_validate**, de mettre à jour un label qui est déclaré dans une autre classe ! Au niveau de InputBox, je n'ai pas accès à cette variable (**self.label_rows** contient une liste de labels).

Il faut que l'évènement **on_text_validate** appelle une fonction qui soit dans une classe parente de **self.label.rows**.
D'une manière générale, je dois donc avoir une classe "super conteneur" qui va contenir les autres conteneurs et widgets. C'est au niveau de cette classe que je déclare les Property, les évènements et les variables "globales" du jeu comme les labels.

Remarque :
Dans Qt, même si on a différents widgets (et classes), ils peuvent communiquer entre eux à l'aide de signaux - des **emit** (qui agissent comme des évènements), que l'on connecte à l'aide de **listeners**, ce qui permet de récupérer des variables de cette manière entre les classes.

Finalement, mon Input_box ressemble à cela :

```
<MyTextInput>:
  size_hint: None, 1
  width: "250dp"
  hint_text: "Votre mot"
  font_size: dp(50)
#  texte validé lorsqu'on appuie sur enter'
  multiline: False
```

et il est situé dans une branche de mon GameLayout :

```
<GameLayout>:

  my_text_input: my_textInput
  my_grid: my_grid
  my_bt_validate: id_bt_validate
  my_bt_replay: id_bt_replay
  my_bt_settings: id_settings

  MainBoxLayout:
    MainGrid:
      id: my_grid
    InputBox:
      MyTextInput:
        id: my_textInput
```

Comme on le voit, c'est au niveau du GameLayout que je récupère tous les chemins vers les composants (Widgets) que je veux utiliser.

Dans le python, on fait le bind sur le TextInput :

```
self.my_text_input.bind(on_text_validate=self.on_text_validate)
```

... pour appeler la méthode principale qui lancera le jeu.

Problème : comment contrôler le contenu du TextInput ?

Solution :

Dans toutes les bonnes bibliothèques graphiques (même dans Tkinter, il existe une méthode de contrôle pour les textInput). Dans kivy, cela existe aussi et il y a des exemples dans la doc.

Cela est implémenté dans le classe du TextInput. Il s'agit de faire un overwrite de la méthode insert_text.

python :

```
class MyTextInput(TextInput):

    nb_cols = NumericProperty()
    pat = re.compile('[^a-z]')

    def insert_text(self, substring, from_undo=False):
        pat = self.pat
        if len(self.text) < self.nb_cols:
            s = re.sub(pat, '', substring)
            s = s.upper()
            return super(MyTextInput, self).insert_text(s, from_undo=from_undo)
        # else:
        #     print("trop grand")
```

Grâce à cette méthode, on affiche uniquement les caractères de a à z (ascii lowercase), que l'on convertit en uppercase, jusqu'à un nombre maximum de caractères (défini par self.nb_cols)

Il existe une deuxième procédure de vérification, qui est appelée lorsqu'on valide le texte (touche entrée ou bouton ok). Cette fonction est définie dans l'api et peut s'adapter à n'importe quel type de validation.

```
def check_word_validation(word, nb_char):
    # On vérifie qu'il y a au max self.NB_COLUMNS - 1 chars dans l'entry
    cond_1 = len(word) == nb_char
    # if not cond_1:
    #     print(f"Le mot ne contient pas {nb_char} caractères")
    # On vérife qu'on ajoute une lettre majuscule
    cond_2 = all([char in string.ascii_uppercase for char in word])
    # if not cond_2:
    #     print(f"Le mot ne contient pas que des lettres majuscules")
    res = cond_1 and cond_2
    return res
```

Beaucoup de lignes sont grisées car c'est une version "pédagogique" et j'ai restreint le nombre de vérifications. En théorie la condition 2 est forcément remplie si le filtre insert_text a bein fait son travail :).

Bonus :

Il existe aussi un évènement lié à l'InputBox lorsqu'on ajoute un caractère.

```
def on_text(self, instance, value):
    print('The widget', instance, 'have:', value)
```

b. Donner le focus au TextInput

Problème : Je veux que l'utilisateur puisse enchainer les propositions de mots sans avoir à cliquer à chaque fois dans le TextInput pour avoir le focus (et le curseur qui clignote).
Naturellement, je mets le focus à True, mais cela ne donne le focus qu'une fraction de seconde.

Solution :

```
def keep_blinking(instance, *args):
    instance.focus = True


def get_focus(text_input):
    Clock.schedule_once(partial(keep_blinking, text_input), .5)
```

Ne me demandez pas pourquoi, mais ça marche.

6) Le son

Problème : jouer des sons

Si j'utilise SoundLoader, tous les sons ne se jouent pas.
C'est comme s'il y avait un temps de chargement avant le play()
Ils se jouent si on lance play() tous les 0,5 s, mais quand c'est plus rapide on n'entend pas tous les sons.

```
from kivy.core.audio import SoundLoader
```

Solution :

Alors j'utilise winsound.PlaySound, qui fonctionne parfaitement (en mode asynchrone)

```
def display_check_word(self, compare_list, *largs):

    letter_code = compare_list[self.read_col]
```

```
    self.my_grid.display_check_letter(self.read_row, self.read_col, letter_code)
    winsound.PlaySound(self.sound_letter[letter_code], winsound.SND_ASYNC)
    # self.sound[letter_code].play()
```

Bonus :

La méthode init_winsound montre comment initialiser les sons avec une compréhension de dict (elle est équivalente aux 3 lignes mises en commentaire au-dessus).

```
def init_winsound(self):
    # WINSOUND
    # self.sound_letter = {}
    # for i in range(3):
    #     self.sound_letter[i] = os.path.join("sounds", f"sound_0{i}.wav")

    self.sound_letter = {x: f"sounds/sound_0{x}.wav" for x in range(3)}
    self.sound_win = "sounds/gagne.wav"
    self.sound_loose = "sounds/perdu.wav"
```

7) La taille de la fenêtre principale (Window)

Problème : La fenêtre principale ne s'adapte pas à la taille des Widgets. Il peut donc y avoir des Widgets qui "débordent", notamment en haut. J'ai remarqué que les Widgets sont collés au bas de la fenêtre (ce qui est en accord avec le système de coordonnées. Mais cela autorise des dépassements vers le droite à le haut.

Solution :

Ce n'est pas l'idéal, mais, dans un premier temps, je mesure les tailles de fenêtre adéquates pour chaque mode de jeu (nb_cols = 5, 6 et 7). Ensuite, je fixe la taille de la fenêtre selon le mode sélectionné par l'utilisateur.

dans le python, au niveau du screenmanager (contrôleur) :

```
def set_window_size(self, nb_cols):
    if nb_cols == 5:
        Window.size = (515, 735)
    elif nb_cols == 6:
        Window.size = (515, 655)
    elif nb_cols == 7:
        Window.size = (515, 595)
```

J'ai choisi de conserver la largeur, parce que voir toutes les dimensions changer donne une mauvaise impression (d'une application buggée)

Bonus :

Pour "prendre les bonnes mesures", j'écoute le changement de Window size :

```
    Window.bind(on_resize=self.track_size)

def track_size(self, instance, width, height):
    print(f"width : {width}")
    print(f"height : {height}")
    print(f"instance : {instance}")
```

Je ne suis pas totalement satisfait car j'aimerais ce ne soit pas la fenêtre principale qui se redimensionne, mais les widgets à l'intérieur (notamment la grid).

Lorsqu'on passe au mode 5 lettres, la grid s'adapte en largeur, et comme on a fixé la hauteur par rapport à la largeur, la grid devient trop haute et peut dépasser de la fenêtre.
On pourrait résoudre ce problème en fixant les dimensions de la grid, mais dans ce cas ce n'est plus du tout responsive.

8) Modifier nb_cols dans la grid suite au choix de l'utilisateur

Problème : comment transmettre la variable nb_cols et comment redessiner la grid ?

Solution :

La structure générale expliquée ci-dessus va nous permettre d'accéder au choix de l'utilisateur.
Tout d'abord, je définis un contrôleur au niveau de MyScreenManager. Normal puisque c'est lui qui contient les screens. Il pourra donc récupérer nb_cols dans le screen_settings et introduire cette valeur dans le screen_game.

```
class MyScreenManager(ScreenManager):
    # Controller

    settings_layout = ObjectProperty()
    game_layout = ObjectProperty()

    screen_settings = ObjectProperty()
    screen_game = ObjectProperty()


    def __init__(self, **kwargs):
        super(MyScreenManager, self).__init__(**kwargs)

        Window.size = (515, 655)

        # first screen
        self.current = self.screen_settings.name

        self.bt_ok = self.settings_layout.bt_ok
```

```
        self.bt_ok.bind(on_press=self.on_press_bt)
        self.game_layout.my_bt_settings.bind(on_press=self.on_settings_press)
```

On remarque que le binding direct ne pose pas de problème ici. C'est parce que settings_layout et game_layout sont initialisés lorsque le init est lancé. Pour cela il faut jeter un coup d'oeil au kv :

```
<MyScreenManager>:
    settings_layout: id_settingsLayout
    game_layout: id_gameLayout
    screen_settings: id_screen_settings
    screen_game: id_screen_game

    Screen:
        id: id_screen_settings
        name: "screen_settings"
        SettingsFloatLayout:
            id: id_settingsLayout
    Screen:
        id: id_screen_game
        name: "screen_game"
        GameLayout:
            id: id_gameLayout
```

Screen étant une instance anonyme, elle ne fait pas obstacle au référencement des layouts enfants (via leur id). Et c'est bien pratique !

Lorsque l'utilisateur valide son choix, on récupère la valeur du counter nb_cols et on fait les opérations nécessaires (ou plutôt commande les opérations nécessaires).

```
def on_press_bt(self, obj):
    # print(f"La valeur est : {self.settings_layout.get_counter()}")
    new_cols = self.settings_layout.get_counter()
    self.current = "screen_game"
    self.transition.direction = 'left'

    self.game_layout.my_grid.repopulate(new_cols)

    self.game_layout.my_text_input.nb_cols = new_cols

    self.game_layout.cols = new_cols

    self.set_window_size(new_cols)

    self.game_layout.init_game()
```

Le dessin de la grid se fait en appelant repopulate.

```
def repopulate(self, cols):

    self.clear_widgets()
    self.cols = cols
    self.labels_rows = []
```

On commence par faire un clear.widgets() qui supprime tous les enfants.
Le tableau de tableaux de labels est vidé. Puis on redessine la grid.

9) Les boutons

a. Les boutons triangles

Problème : Je veux des boutons triangles de part et d'autre de nb_col choisi par l'utilisateur, afin d'incrémenter ou décrémenter le nb_col.
- Il n'existe pas de bouton triangle. Par défaut, les boutons sont rectangulaires
- On va avoir un problème de zone cliquable, puisque cette zone est rectangulaire par défaut, peu importe le canva.

Solution :

Pour le dessin du triangle, ce n'est pas compliqué.

```
<TriangleButtonRight>:

  size_hint: None, None
  size: dp(50), dp(50)

  canvas.before:
    Color:
      rgb: hex(cst.color_bleu_3) if self.state == 'normal' else hex(cst.violet_1)
    Triangle:
      points:self.x,self.y,self.x,self.y + self.height,self.x + self.width,self.y + self.height/2
    Color:
      rgb: hex(cst.color_orange)
    Line:
      width: 2
      close: True
      points:self.x,self.y, self.x,self.y + self.height, self.x + self.width,self.y + self.height/2
```

On ajoute une Line pour avoir un contour.
On définit une couleur pour l'état normal et une autre pour le state "down" (button clicked)

Bonus :

Pour l'alignement, la solution la plus simple et efficace que j'ai trouvée consiste à placer le Widget dans un RelativeLayout :

```
RelativeLayout:
  TriangleButtonLeft:
    id: bt_backward
    pos_hint: {'center_x': 0.5, 'center_y': 0.5}
```

La partie la plus compliquée concerne la zone cliquable.
Il s'agit de redéfinir la méthode **collide_point**.

Après quelques recherches et avoir trouvé des équations incompréhensibles, j'ai fait un dessin et opté pour un raisonnement personnel :

La zone cliquable est l'intersection de 3 zones, définies chacune par un côté du triangle. Je cherche donc l'équation de chaque droite correspondante.

```
class TriangleButtonRight(ButtonBehavior, Widget):
    def collide_point(self, x, y):
        x0 = self.x
        y0 = self.y
        x1 = x0 + self.width
        y1 = y0 + self.height / 2
        y2 = y0 + self.height
        a1 = (y0 - y1) / (x0 - x1)
        a2 = (y2 - y1) / (x0 - x1)

        collide_inf = False
        collide_border = False
        collide_sup = False

        if y >= a1 * x + y0 - x0 * a1:
            collide_inf = True

        if x >= x0:
            collide_border = True

        if y <= a2 * x + y2 - a2 * x0:
            collide_sup = True

        return all([collide_inf, collide_border, collide_sup])
```

$$\begin{cases} y_1 = a x_1 + b & (1) \\ y_2 = a x_0 + b & (2) \end{cases}$$

$y_2 - y_1 = a(x_0 - x_1)$

$\Rightarrow a = \dfrac{y_2 - y_1}{x_0 - x_1}$

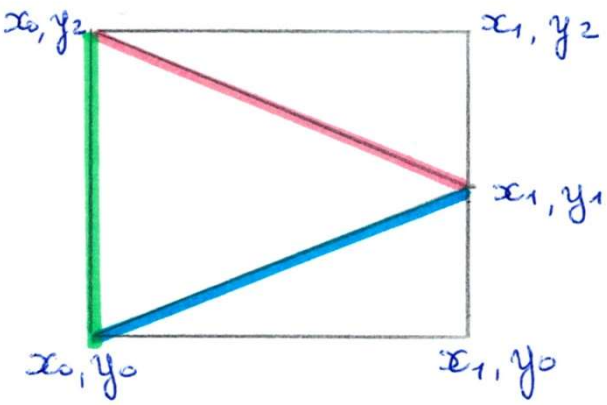
(2) $b = y_2 - a x_0$

La droite a pour équation :

$y = a x + y_2 - a x_0$

où $a = \dfrac{y_2 - y_1}{x_0 - x_1}$

triangle de droite :



$x = x_0$

$$\begin{cases} y_0 = a x_0 + b & (1) \\ y_1 = a x_1 + b & (2) \end{cases}$$

$y_0 - y_1 = a x_0 - a x_1$

$= a(x_0 - x_1)$

$\Rightarrow a = \dfrac{y_0 - y_1}{x_0 - x_1}$

(1) $b = y_0 - a x_0$

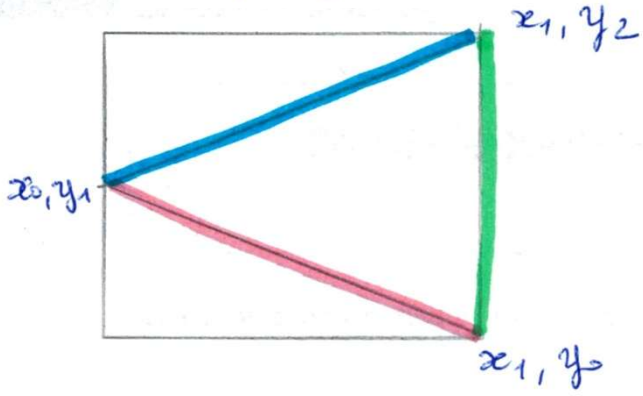
On peut écrire l'équation de la droite : $y = a x + y_0 - a x_0$

où $a = \dfrac{y_0 - y_1}{x_0 - x_1}$

(x_0, y_0) du triangle de droite (x_0, y_1) correspond à $\longrightarrow$

(x_1, y_1) $\longrightarrow$ (x_1, y_2)

triangle gauche



$x = x_1$

(x_0, y_2) $\longrightarrow$ (x_0, y_1)

(x_1, y_1) $\longrightarrow$ (x_1, y_0)

b. Les boutons rounded corner

Problème : Je veux des boutons rectangulaires aux bords arrondis avec une bordure.
On ne peut pas faire de **border_color** car la zone du bouton est toujours rectangulaire.

Solution :

```
<CustomButton@Button>:
  pos_hint: {"center_x":.5, "center_y":.5}
  background_color: (0, 0, 0, 0)
  # transparent
  border_radius: [10]
  border_radius_line: 10
  back_color: (1, 0, 1, 0)
  border_color: (0, 0, 0, .5)
  border_width: 1.5
  canvas.before:
    Color:
      rgba: self.back_color
    RoundedRectangle:
      size: self.size
      pos: self.pos
      radius: self.border_radius
    Color:
      rgba: self.border_color
    Line:
      width: self.border_width
      rounded_rectangle: (self.pos[0], self.pos[1], self.size[0], self.size[1], self.border_radius_line)
```

On définit 2 couleurs : back_color pour la couleur de fond du bouton et border_color pour la couleur de la bordure.

Pour colorer le fond, j'utilise un RoundedRectangle, tandis que la bordure est une Line de rounded_rectangle. Notez les valeurs pour les radius qui sont sous forme de liste pour le Rounded_Rectangle et d'un simple integer pour la ligne.
Enfin, un background_color transparent (alpha = 0) s'applique au bouton (rectangulaire) afin que l'on ne voit pas le bouton sous notre dessin dans le canva (sinon on verrait les angles).

Je personnalise le bouton comme cela (par exemple dans le settings screen :

```
CustomButton:
  id: bt_ok
  text: "OK"
  bold: True
  font_size: dp(30)
  size_hint: None, None
  size: dp(100), dp(50)
  back_color: hex(cst.color_orange) if self.state == 'normal' else hex(cst.color_orange_f)
```