CASSE BRIQUE

Lucas Willems
Aymeric Lhullier

Table des matières :

Introduction :

Nous avons choisi de réaliser un casse brique pour notre projet. Seront alors exposés, dans ce dossier, les différents éléments relatifs à la réalisation du programme, soit :

- Un cahier des charges présentant les différentes contraintes/besoins à traiter
- Un rapport technique comprenant quelques parties du programme écrites en algorithme, une copie d'écran d'une exécution du programme, les explications de ce programme, l'exposition des difficultés rencontrées et des solutions trouvées
- Un manuel d'utilisation expliquant comment utiliser le programme
- Une annexe contenant les programmes dans le langage Python

Cahier des charges :

Notre casse brique doit posséder :

- Les fonctionnalités de base :
  - gestion des collisions entre balle, barre et briques
  - gestion du rebond sur la barre : changement de direction selon l'endroit où elle rebondit
- Des fonctionnalités supplémentaires :
  - un système très modulable de niveaux
  - un système de score qui correspond au temps mis par l'utilisateur pour faire tous les niveaux
  - une prise en compte de divers effets avec des briques qui permettent d'obtenir des actions spéciales une fois cassées
  - un éditeur de niveau pour créer très facilement ses propres niveaux

Rapport technique

Notre casse brique est en réalité composé de 2 programmes :

- `game.py` programme pour jouer au Casse brique
- `editor.py` programme pour créer/éditer des niveaux pour jouer au Casse brique

Dans les grandes lignes, voilà comment `game.py` fonctionne :

1. Création de la fenêtre, de la barre, de la balle, du chronomètre…
2. Remise à zéro des composants cités : la barre est placée en bas au centre, la balle est placée sur la balle…
3. Chargement du niveau 1 : lecture du fichier correspondant au niveau et affichage des briques à l'écran

4. Début du jeu :
    - les entrées clavier sont prises en compte : le joueur peut déplacer la barre avec les flèches directionnelles ou appuyer sur espace pour lancer la balle
    - une fonction nextFrame, appelée en permanence (toutes les 1/60ème de seconde) :
        - détecte si la balle est en collision avec une brique ou avec la barre ou si elle est sortie de l'écran, puis, calcule sa nouvelle position
        - détecte si le joueur a gagné (toutes les briques ont été cassées) ou perdu (la balle n'a pas été touchée par la barre), puis affiche un message en fonction, et enfin charge le nouveau niveau en ayant remis à zéro tous les composants au préalable (étape 2)

Toujours dans les grandes lignes, voici commet editor.py fonctionne :

1. Saisie du niveau X à créer/éditer
2. Création de la fenêtre et chargement du niveau : les "." sont convertis en briques blanches
3. Détection de la souris :
    - Clique gauche permet de sélectionner une couleur de brique (en cliquant sur les briques en bas à gauche) puis de remplacer les briques du niveau par la couleur de la brique sélectionnée (en cliquant sur les briques du niveau)
    - Clique droit permet de supprimer une brique de niveau, c'est-à-dire de la remplacer par une blanche
4. Mise à jour automatique du fichier à chaque modification du niveau : les briques blanches sont remplacées par des "."

Notre façon de procéder venant d'être explicitée, voici nos réalisations :

- Un algorithme sur les collisions (réalisé par Aymeric)

```
#Obtention de la position relative de 2 objets pour savoir s'ils sont entrés en collision
#Entrée : "el1" et "el2" 2 élements graphiques définis par des coordonnées
#Sortie : "collisionCounter", un entier qui indique la position du premier element par rapport au deuxieme

Algorithme collision(el1, el2) :
        #Initialisation de la variable de sortie à zéro
        collisionCounter = 0

        #Récupération des coordonnées des objets dans des listes du type (pour un rectangle) :
        #[x du coin superieur gauche, y du coin superieur gauche, x du coin inferieur droit, y du coin inferieur
droit]
        el1 = coordonnées de el1
        el2 = coordonnées de el2

        #Réalisation des tests suivants :
        #Est-ce que l'objet, par rapport à l'obstacle, se trouve...
```

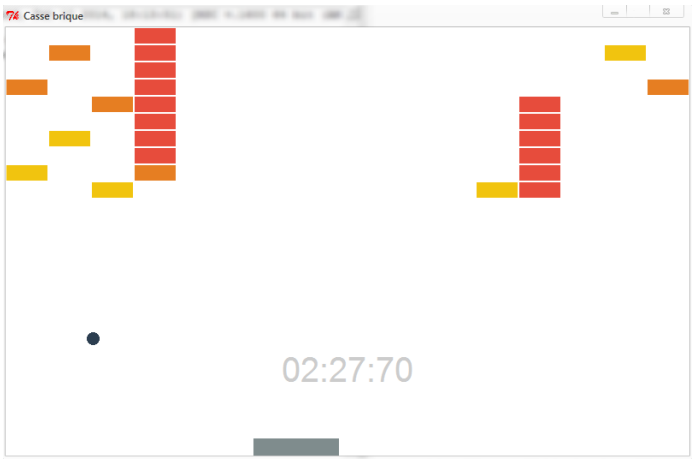
```
        #à gauche
        si el1[2] < el2[0]
                alors collisionCounter = 1
        #en dessous
        si el1[3] < el2[1]
                alors collisionCounter = 2
        #à droite
        si el1[0] > el2[2]
                alors collisionCounter = 3
        #au dessus
        si el1[1] > el2[3]
                alors collisionCounter = 4

        #collisionCounter stocke la position de el1 par rapport à el2
        #Si collisionCounter vaut 0 (el1 n'est ni au dessus, ni à droite, ni à gauche, ni au dessous de el2),
        #alors il y a collision

        retourner collisionCounter
```

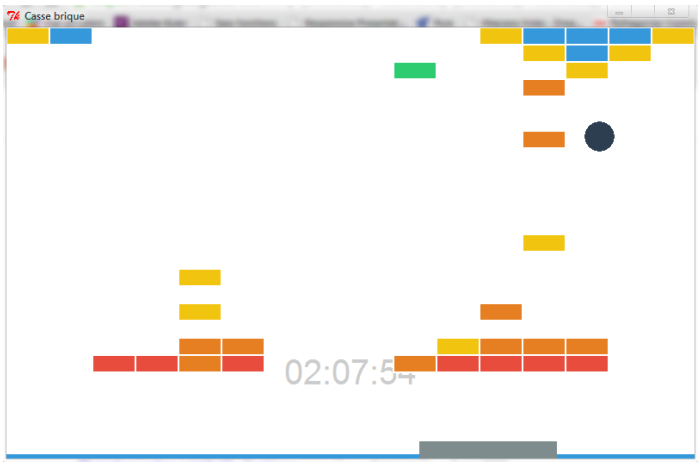
- Un algorithme sur le rebond de la balle sur la barre (réalisé par Lucas)

```
#Calcul l'angle après rebond de la balle sur la barre en fonction de l'endroit où elle a rebondi
#Entrée : balleX, un entier qui est la coordonnée en x du centre de la balle, balleAngle, un entier qui est l'angle avant rebond de la balle, barreX, un entier qui est la coordonnée en x du centre de la barre, et barreLargeur, un entier qui est la largeur de la balle
#Sortie : angleFinal, un entier qui est l'angle de la balle après rebond

Algorithme rebond(balleX, balleAngle, barreX, barreLargeur):
        #La différence de X entre le centre de la balle et le centre de la barre
        diffX = balleX - barreX

        #L'angle de la balle après rebond non déformé en prenant en compte l'angle avant rebond
        angleNormal = (-balleAngle) % 2pi

        #L'angle de la balle après rebond déformé sans prendre en compte l'angle avant rebond
        #Angle obtenu grâce à une équation de droite obtenue en posant :
        #angleCalculé = 20° si diffX = barreLargeur/2
        #angleCalculé = 160° si diffX = -barreLargeur/2
        angleCalculé = -70/(barreLargeur/2)*diffX + 90

        #L'angle final de la balle après rebond en parti déformé
        #Angle obtenu en posant :
        #angleFinal = angleNormal si abs(diffX) = 0
        #angleFinal = angleCalculé si abs(diffX) = barreLargeur/2
        angleFinal = (1 - (abs(diffX)/(barreLargeur/2))**0.25)*angleNormal +
((abs(diffX)/(barreLargeur/2))**0.25)*angleCalculé

        retourner angleFinal
```

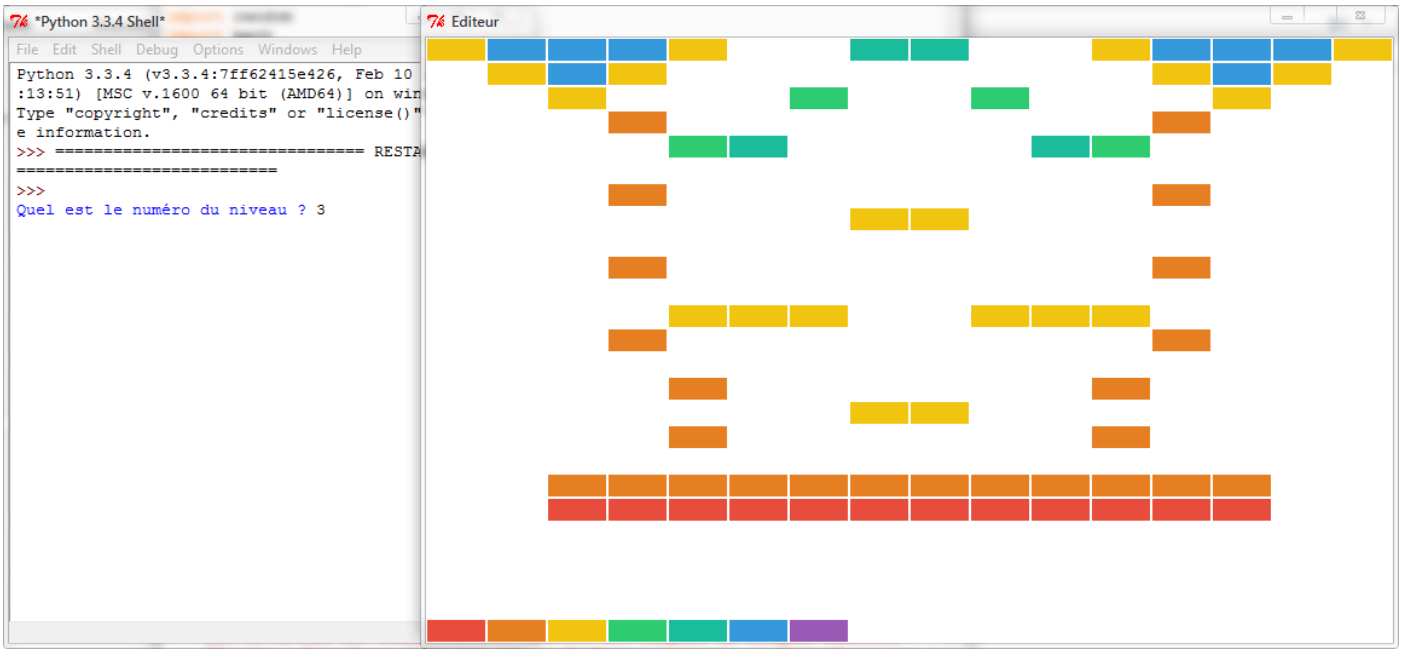
- Plusieurs captures d'écran :

game.py





editor.py



Pour réaliser cet algorithme et programme, nous avons rencontré plusieurs problèmes :

- Problème : briques qui restent à l'écran après avoir été cassées
  Solution : supprimer toutes les briques se trouvant à l'écran à chaque fois qu'un niveau est relancé
- Problème : balle qui reste coincé dans un coin de l'écran quand elle se déplace trop vite
  Solution : faire changer la balle de direction juste avant qu'elle sorte de l'écran

Manuel d'utilisation

Pour utiliser game.py, il faut ouvrir le programme dans Python et appuyer sur F5. Il ne suffit plus qu'à utiliser la barre d'espace pour lancer la balle et les flèches directionnelles pour déplacer la barre.

Pour utiliser editor.py, il faut ouvrir le programme dans Python et appuyer sur F5. Ensuite, il faut saisir le numéro du niveau que l'on veut éditer/créer. Il ne reste plus qu'à sélectionner (clique gauche) les briques en bas à droite et à les mettre dans le niveau (clique gauche). Il est aussi possible de supprimer des briques du niveau (clique droit).

Conclusion

Pour conclure, ce casse brique est un vrai petit jeu, assez complet, qui implémente de nombreux concepts du monde du jeu : collisions, système de score…

La réalisation de ce jeu nous a aussi permis de connaître plus en profondeur Python, de découvrir Tkinter, de créer de nouveaux algorithmes (rebond de la balle sur la barre) et de nous forcer à concevoir notre programme de la manière la plus claire possible.

Annexe

Le code source du programme game.py :

```
import tkinter as tk
import random
import math
import copy

#Classe principale du programme : contient les fonctions(methodes) et les variables(attributs) qui vont servir à faire fonctionner le programme
#Elle "hérite" de tkinter et peut donc se servir des methodes et attribus de tkinter
class Area(tk.Canvas):

   ###Variable(attributs)###

   barHeight = 20
   barSpeed = 10

   ballSpeed = 7

   bricks = []
   bricksWidth = 50
   bricksHeight = 20
   bricksNb = 16
   linesNb = 20
   bricksColors = {
      "r": "#e74c3c",
      "g": "#2ecc71",
      "b": "#3498db",
      "t": "#1abc9c",
      "p": "#9b59b6",
      "y": "#f1c40f",
      "o": "#e67e22",
   }

   textDisplayed = False

   screenHeight = 500
   screenWidth = bricksWidth*bricksNb

   seconds = 0

   ###Fonction(methodes)###

   #Cette méthode, appelée à l'initialisation de la classe (appelée au debut du programme),
   #initialise divers attributs nécessaires au programme (la bar, la balle...)
   #Donnée : "root" un objet qui est la fenêtre principale
   def __init__(self, root):
      tk.Canvas.__init__(self, root, bg="#ffffff", bd=0, highlightthickness=0, relief="ridge", width=self.screenWidth, height=self.screenHeight)
      self.pack()
```

```
      self.timeContainer = self.create_text(self.screenWidth/2, self.screenHeight*4/5, text="00:00:00", fill="#cccccc",
font=("Arial", 30), justify="center")
      self.shield = self.create_rectangle(0, 0, 0, 0, width=0)
      self.bar = self.create_rectangle(0, 0, 0, 0, fill="#7f8c8d", width=0)
      self.ball = self.create_oval(0, 0, 0, 0, width=0)
      self.ballNext = self.create_oval(0, 0, 0, 0, width=0, state="hidden")
      self.level(3)
      self.nextFrame()

   #Cette méthode, appelée à chaque fois qu'un niveau est rechargé (ou recommencé),
   #remet à zéro tous les éléments (taille et position de la barre...)
   def reset(self):
      self.barWidth = 100
      self.ballRadius = 7
      self.coords(self.shield, (0, self.screenHeight-5, self.screenWidth, self.screenHeight))
      self.itemconfig(self.shield, fill=self.bricksColors["b"], state="hidden")
      self.coords(self.bar, ((self.screenWidth - self.barWidth)/2, self.screenHeight - self.barHeight, (self.screenWidth +
self.barWidth)/2, self.screenHeight))
      self.coords(self.ball, (self.screenWidth/2 - self.ballRadius, self.screenHeight - self.barHeight - 2*self.ballRadius,
self.screenWidth/2 + self.ballRadius, self.screenHeight - self.barHeight))
      self.itemconfig(self.ball, fill="#2c3e50")
      self.coords(self.ballNext, tk._flatten(self.coords(self.ball)))
      self.effects = {
         "ballFire": [0, 0],
         "barTall": [0, 0],
         "ballTall": [0, 0],
         "shield": [0, -1],
      }
      self.effectsPrev = copy.deepcopy(self.effects)
      self.ballThrown = False
      self.keyPressed = [False, False]
      self.losed = False
      self.won = False
      self.ballAngle = math.radians(90)
      for brick in self.bricks:
         self.delete(brick)
         del brick

   #Cette méthode est chargée de "créer" le niveau X en lisant le fichier correspondant ("X.txt")
   #Elle affiche 1 à 1 les briques qui formeront le niveau
   #Donnée : "level" un entier qui le numéro du niveau
   def level(self, level):
      self.reset()
      self.levelNum = level
      self.bricks = []
      try:
         file = open(str(level)+".txt")
         content = list(file.read().replace("\n", ""))[:(self.bricksNb*self.linesNb)]
         file.close()
         for i, el in enumerate(content):
            col = i%self.bricksNb
            line = i//self.bricksNb
            if el != ".":
```

```
                self.bricks.append(self.create_rectangle(col*self.bricksWidth, line*self.bricksHeight,
(col+1)*self.bricksWidth, (line+1)*self.bricksHeight, fill=self.bricksColors[el], width=2, outline="#ffffff"))
        #S'il n'y a plus de niveau à charger, le jeu est fini et on affiche l'écran de fin du jeu (avec le temps du joueur)
        except IOError:
            self.displayText("JEU FINI EN\n" + "%02d mn %02d sec %02d" % (int(self.seconds)//60, int(self.seconds)%60,
(self.seconds*100)%100), hide = False)
            return
        self.displayText("NIVEAU\n"+str(self.levelNum))

    #Cette méthode, appelée en boucle (tous les 1/60 de seconde),
    #recalcule les coordonnées de tous les éléments (déplacement de la balles, collisions, mise à jour des effets)
    def nextFrame(self):
        if self.ballThrown and not(self.textDisplayed):
            self.moveBall()

        if not(self.textDisplayed):
            self.updateTime()

        self.updateEffects()

        if self.keyPressed[0]:
            self.moveBar(-area.barSpeed)
        elif self.keyPressed[1]:
            self.moveBar(area.barSpeed)

        if not(self.textDisplayed):
            if self.won:
                self.displayText("GAGNE !", callback = lambda: self.level(self.levelNum+1))
            elif self.losed:
                self.displayText("PERDU !", callback = lambda: self.level(self.levelNum))

        self.after(int(1000/60), self.nextFrame) #au bout de 1000/60 ms on rappelle la fonction (il y a donc 60 frames
par seconde)

    #Cette méthode, appelée à l'appui sur les flèches de gauche et de droite,
    #est utilisée pour déplacer la barre de "x" pixels horizontalement en vérifiant qu'elle ne sorte pas de l'écran
    #Si la balle n'est pas encore lancée, la méthode se charge aussi de la deplacer en même temps que la barre
    #Donnée : "x" un entier qui correspond au déplacement en X en nombre de pixels
    def moveBar(self, x):
        barCoords = self.coords(self.bar)
        if barCoords[0] < 10 and x < 0:
            x = -barCoords[0]
        elif barCoords[2] > self.screenWidth - 10 and x > 0:
            x = self.screenWidth - barCoords[2]

        self.move(self.bar, x, 0)
        if not(self.ballThrown):
            self.move(self.ball, x, 0)

    #Cette methode, appelée à chaque frame, est utilisée pour déplacer la balle
    #Elle calcule les collisions entre la balle et les briques/barre/bord de l'écran
    #ainsi que le déplacement de la balle en se servent des attributs "ballAngle" et "ballSpeed"
    #Elle applique aussi les effets à la balle et à la barre lors des collisions avec des briques speciales
    def moveBall(self):
```

```
        self.move(self.ballNext, self.ballSpeed*math.cos(self.ballAngle), -self.ballSpeed*math.sin(self.ballAngle))
        ballNextCoords = self.coords(self.ballNext)

        #Calcul des collisions entre la balle et les briques
        i = 0
        while i < len(self.bricks):
            collision = self.collision(self.ball, self.bricks[i])
            collisionNext = self.collision(self.ballNext, self.bricks[i])
            if not collisionNext:
                brickColor = self.itemcget(self.bricks[i], "fill")
                if brickColor == self.bricksColors["g"]:   #effet "barTall" (brique verte)
                    self.effects["barTall"][0] = 1
                    self.effects["barTall"][1] = 240
                elif brickColor == self.bricksColors["b"]: #effet "shield" (brique bleu)
                    self.effects["shield"][0] = 1
                elif brickColor == self.bricksColors["p"]: #effet "ballFire" (brique violette)
                    self.effects["ballFire"][0] += 1
                    self.effects["ballFire"][1] = 240
                elif brickColor == self.bricksColors["t"]: #effet "ballTall" (brique turquoise)
                    self.effects["ballTall"][0] = 1
                    self.effects["ballTall"][1] = 240

                if not(self.effects["ballFire"][0]):
                    if collision == 1 or collision == 3:
                        self.ballAngle = math.radians(180) - self.ballAngle
                    if collision == 2 or collision == 4:
                        self.ballAngle = -self.ballAngle

                if brickColor == self.bricksColors["r"] :   #si la bricks est rouge, elle devient orange
                    self.itemconfig(self.bricks[i], fill=self.bricksColors["o"])
                elif brickColor == self.bricksColors["o"] : #si la bricks est orange, elle devient jaune
                    self.itemconfig(self.bricks[i], fill=self.bricksColors["y"])
                else:                               #si la bricks est jaune (ou toute autre couleur ormis rouge/orange), elle est
detruite
                    self.delete(self.bricks[i])
                    del self.bricks[i]
            i += 1

        self.won = len(self.bricks) == 0

        #Calcul des collisions entre la balle et les bords de l'ecran
        if ballNextCoords[0] < 0 or ballNextCoords[2] > self.screenWidth:
            self.ballAngle = math.radians(180) - self.ballAngle
        elif ballNextCoords[1] < 0:
            self.ballAngle = -self.ballAngle
        elif not(self.collision(self.ballNext, self.bar)):
            ballCenter = self.coords(self.ball)[0] + self.ballRadius
            barCenter = self.coords(self.bar)[0] + self.barWidth/2
            angleX = ballCenter - barCenter
            angleOrigin = (-self.ballAngle) % (3.14159*2)
            angleComputed = math.radians(-70/(self.barWidth/2)*angleX + 90)
            self.ballAngle = (1 - (abs(angleX)/(self.barWidth/2))**0.25)*angleOrigin +
((abs(angleX)/(self.barWidth/2))**0.25)*angleComputed
        #Si la balle touche le bas de l'ecran, on a perdu, sauf si on dispose de l'effet "shield"
```

```
      elif not(self.collision(self.ballNext, self.shield)):
         if self.effects["shield"][0]:
            self.ballAngle = -self.ballAngle
            self.effects["shield"][0] = 0
         else :
            self.losed = True

      self.move(self.ball, self.ballSpeed*math.cos(self.ballAngle), -self.ballSpeed*math.sin(self.ballAngle))
      self.coords(self.ballNext, tk._flatten(self.coords(self.ball)))

   #Cette méthode, appelée à chaque frame, gère le temps restant pour chacun des effets et les affichent (taille de la barre, balle...)
   def updateEffects(self):
      for key in self.effects.keys():
         if self.effects[key][1] > 0:
            self.effects[key][1] -= 1
         if self.effects[key][1] == 0:
            self.effects[key][0] = 0

      if self.effects["ballFire"][0]: #l'effet "ballFire" permet à la balle de detruire les briques sans rebondir dessus (elle les traverse)
         self.itemconfig(self.ball, fill=self.bricksColors["p"])
      else:
         self.itemconfig(self.ball, fill="#2c3e50")

      if self.effects["barTall"][0] != self.effectsPrev["barTall"][0]: #l'effet "barTall" agrandit la taille de la bar
         diff = self.effects["barTall"][0] - self.effectsPrev["barTall"][0]
         self.barWidth += diff*60
         coords = self.coords(self.bar)
         self.coords(self.bar, tk._flatten((coords[0]-diff*30, coords[1], coords[2]+diff*30, coords[3])))
      if self.effects["ballTall"][0] != self.effectsPrev["ballTall"][0]: #l'effet "ballTall" agrandit la taille de la balle
         diff = self.effects["ballTall"][0] - self.effectsPrev["ballTall"][0]
         self.ballRadius += diff*10
         coords = self.coords(self.ball)
         self.coords(self.ball, tk._flatten((coords[0]-diff*10, coords[1]-diff*10, coords[2]+diff*10, coords[3]+diff*10)))

      if self.effects["shield"][0]: #l'effet "shield" permet à la balle de rebondir 1 fois sur le bas de l'ecran (c'est une "vie" supplementaire)
         self.itemconfig(self.shield, fill=self.bricksColors["b"], state="normal")
      else:
         self.itemconfig(self.shield, state="hidden")

      self.effectsPrev = copy.deepcopy(self.effects)

   #Cette méthode met à jour le temps de jeu (affiché en fond)
   #Les minutes, les secondes et les centiemes de seconde sont affichés
   def updateTime(self):
      self.seconds += 1/60
      self.itemconfig(self.timeContainer, text="%02d:%02d:%02d" % (int(self.seconds)//60, int(self.seconds)%60, (self.seconds*100)%100))

   #Cette méthode permet d'afficher du texte sur l'ecran
   #Donnée : "text" le texte à afficher, "hide" un bouléen pour définir si le texte doit être caché au bout d'un certain temps, "callback" une fonction appelée (si définie) au bout de ce même certain temps
```

```
    def displayText(self, text, hide = True, callback = None):
        self.textDisplayed = True
        self.textContainer = self.create_rectangle(0, 0, self.screenWidth, self.screenHeight, fill="#ffffff", width=0,
stipple="gray50")
        self.text = self.create_text(self.screenWidth/2, self.screenHeight/2, text=text, font=("Arial", 25), justify="center")
        if hide:
            self.after(3000, self.hideText)
        if callback != None:
            self.after(3000, callback)

    #Cette methode permet de supprimer le texte qui est affiché à l'ecran
    def hideText(self):
        self.textDisplayed = False
        self.delete(self.textContainer)
        self.delete(self.text)

    #Cette methode permet de caculer la position relative entre 2 objets et donc les collisions
    #Donnée : "el1" et "el2" 2 élements graphiques définis des coordonnées
    def collision(self, el1, el2):
        collisionCounter = 0

        objectCoords = self.coords(el1)
        obstacleCoords = self.coords(el2)

        if objectCoords[2] < obstacleCoords[0] + 5:
            collisionCounter = 1
        if objectCoords[3] < obstacleCoords[1] + 5:
            collisionCounter = 2
        if objectCoords[0] > obstacleCoords[2] - 5:
            collisionCounter = 3
        if objectCoords[1] > obstacleCoords[3] - 5:
            collisionCounter = 4

        return collisionCounter

###FIN DE LA CLASSE###


#Fonction appelée si une touche est enfoncée
#Donnée : "event" l'évènement
def eventsPress(event):
    global area, hasEvent

    if event.keysym == "Left":
        area.keyPressed[0] = 1
    elif event.keysym == "Right":
        area.keyPressed[1] = 1
    elif event.keysym == "space" and not(area.textDisplayed):
        area.ballThrown = True

#Fonction appelée si une touche est relachée
#Donnée : "event" l'évènement
def eventsRelease(event):
    global area, hasEvent
```

```
    if event.keysym == "Left":
        area.keyPressed[0] = 0
    elif event.keysym == "Right":
        area.keyPressed[1] = 0

###PROGRAMME PRINCIPAL###

#Le programme principal est tres court car le jeu est surtout géré par la classe
#Dans ces quelques lignes, on créé la fenetre de jeu et on appelle la classe Area
#On "initialise" egalement le clavier (pour que le programme détecte les entrées clavier)
root = tk.Tk()
root.title("Casse brique")
root.resizable(0,0)
root.bind("<Key>", eventsPress)
root.bind("<KeyRelease>", eventsRelease)

area = Area(root)
root.mainloop()
```

Le code source du programme editor.py :

```
import tkinter as tk
import random
import math
import copy

#Classe principale du programme : contient les fonctions(methodes) et les variables(attributs) qui vont servir à faire fonctionner le programme
#Elle "hérite" de tkinter et peut donc se servir des methodes et attribus de tkinter
class Area(tk.Canvas):

    ###Variable(attributs)###

    bricksWidth = 50
    bricksHeight = 20
    bricksNb = 16
    linesNb = 20
    bricksColors = {
        "r": "#e74c3c",
        "g": "#2ecc71",
        "b": "#3498db",
        "t": "#1abc9c",
        "p": "#9b59b6",
        "y": "#f1c40f",
        "o": "#e67e22",
    }

    screenHeight = 500
    screenWidth = bricksWidth*bricksNb

    ###Fonction(methodes)###
```

```
   #Cette méthode, appelée à l'initialisation de la classe (appelée au debut du programme), commence par créer la fenêtre.
   #Si le fichier "X.txt" (avec X le numero du niveau) est existant,
   #les briques du niveau sont placées dans la fenêtre où les "." sont remplacés par des briques blanches
   #Si le niveau est inexistant, la fenêtre est remplie de briques blanches
   #Donnée : "root" un objet qui est la fenêtre principale et "level" qui est le numéro du niveau à ouvrir
   def __init__(self, root, level):
      tk.Canvas.__init__(self, root, bg="#ffffff", bd=0, highlightthickness=0, relief="ridge", width=self.screenWidth, height=self.screenHeight)
      self.level = level
      try:
         file = open(str(self.level)+".txt")
         bricks = list(file.read().replace("\n", ""))[:(self.bricksNb*self.linesNb)]
         file.close()
      except IOError:
         bricks = []
      for i in range(self.bricksNb*self.linesNb-len(bricks)):
         bricks.append(".")
      for i, j in enumerate(bricks):
         col = i%self.bricksNb
         line = i//self.bricksNb
         if j == ".":
            color = "#ffffff"
         else:
            color = self.bricksColors[j]
         self.create_rectangle(col*self.bricksWidth, line*self.bricksHeight, (col+1)*self.bricksWidth, (line+1)*self.bricksHeight, fill=color, width=2, outline="#ffffff")
      for i, j in enumerate(self.bricksColors.items()):
         self.create_rectangle(i*self.bricksWidth, self.screenHeight-self.bricksHeight, (i+1)*self.bricksWidth, self.screenHeight, fill=j[1], width=2, outline="#ffffff")
      self.pack()

   #Cette methode, appelée à chaque fois que l'on veut changer la couleur d'une brique,
   #change la couleur de la brique sur l'écran et enregistre la nouvelle grille dans le fichier "X.txt" (avec X le numero du niveau)
   #où les briques blanches sont remplacées par des "."
   #Donnée : "id" qui est l'identiant (un entier unique) de l'élément et "color" qui est la couleur de la brique séléctionnée
   def setColor(self, id, color):
      self.itemconfig(id, fill=color)

      content = ""
      for i in range(self.bricksNb*self.linesNb):
         if i%self.bricksNb == 0 and i != 0:
            content += "\n"
         brickColor = self.itemcget(i+1, "fill")
         brickId = [id for id, color in self.bricksColors.items() if color == brickColor]
         if brickId == []:
            content += "."
         else:
            content += brickId[0]

      file = open(str(self.level)+".txt", "w")
      file.write(content)
```

```
        file.close()

###FIN DE LA CLASSE###


#Fonction appelée lorsque l'utilisateur appuit sur le bouton gauche de la souris
#Elle rend la brique qui se trouve à l'endroit du curseur de la couleur de la brique sélectionnée
#ou, si le cuseur se trouve sur une des briques en bas de l'écran, elle selectionne une couleur
#Donnée : "event" l'évènement
def eventsLeftClick(event):
    global area

    id = event.widget.find_closest(event.x, event.y)[0]
    if id <= area.bricksNb*area.linesNb:
        if hasattr(area, "selectedColor"):
            area.setColor(id, area.selectedColor)
    else:
        area.selectedColor = area.itemcget(id, "fill")

#Fonction appelée lorsque l'utilistaeur appuit sur le bouton droit de la souris
#Elle rend blanche la brique qui se trouve à l'endroit du curseur
#Donnée : "event" l'évènement
def eventsRightClick(event):
    global area

    id = event.widget.find_closest(event.x, event.y)[0]
    if id <= area.bricksNb*area.linesNb:
        area.setColor(id, "#ffffff")

###PROGRAMME PRINCIPAL###

#Le programme principal est tres court car l'éditeur est surtout géré par la classe
#Dans ces quelques lignes, on créé la fenetre de l'éditeur et on appelle la classe Area
#On "initialise" egalement la souris (pour que le programme détecte les entrées souris)
#On demande aussi à l'utilisateur le numero du niveau qu'il veut créer ou éditer
root = tk.Tk()
root.title("Editeur")
root.resizable(0,0)
root.bind("<Button-1>", eventsLeftClick)
root.bind("<Button-3>", eventsRightClick)

area = Area(root, int(input("Quel est le numéro du niveau ? ")))
root.mainloop()
```