Chapter 4

グラフィクスとイベント処理

4.1 イベント処理

人間と計算機との対話を実現するインタラクティブ (対話的) なプログラムでは，人間の入力に応じてプログラムの動作が変わる．特にグラフィクスを用いたユーザインタフェースを利用する場合には，マウスやキーボードからのユーザ入力によって，プログラムが振る舞いを変更することになる．このユーザ入力のように，いつ生じるかわからない状態変更に対応する手法として，イベントの考え方が用いられる．

イベント (event) とは，ユーザ入力などによって生じる状態変更のことで，マウスのボタンをプレスないしリリースしたり，キーボードから文字を入力したりすると，状態の変更，すなわちイベントが発生したものとみなされる．グラフィクスを用いた対話的なプログラムでは，イベントの発生を受けて，別の作業 (Python の場合には関数) へと実行が移る．このようにイベントの発生に応じてプログラムの実行が変化する方式を，イベント駆動型 (event driven) のプログラムと呼ぶ．イベント発生に応じて実行されるプログラム，すなわち関数ないしメソッドは，コールバック関数 (callback function) ないしコールバックメソッドと呼ばれ，イベントごとに予め登録しておく必要がある．

4.2 tkinter におけるイベント処理

tkinter における入力イベント処理は，Canvas クラスからのコールバック関数によって実現されている．

Canvas クラス： 描画領域のクラス

Canvas オブジェクト内で入力イベントが発生すると，Canvas オブジェクトに登録されたコールバック関数が呼び出され，コールバック関数内でイベント処理が進められる．

bind　コールバック関数登録メソッド

MouseEvent に付随する修飾子を獲得する．修飾子からイベントが発生したボタンの種類 (左中右) や，そのときのキーボードの状態などを調べられる．

なお MouseEvent に関するイベントの種類と名称は，次の表のようになっている．また，図 4.1 は bind によるコールバック関数の登録と，イベント発生時のプログラム実行の遷移を示している．

イベントの種類	イベントの名称 (bind メソッドで利用)
ボタン n のプレス	<Button-n>
ボタン n のリリース	<ButtonRelease-n>
ボタン n でのドラッグ	<Bn-Motion>
カーソルの進入	<Enter>
カーソルの退出	<Leave>

Event クラス： イベントのクラス Event

発生したイベントに関する情報をまとめたクラス．コールバック関数に渡される．

x　イベント発生時の x 座標値
　マウスイベントが発生した際にマウスカーソルがあった場所の x 座標値を返す.

y　イベント発生時の y 座標値
　マウスイベントが発生した際にマウスカーソルがあった場所の y 座標値を返す.

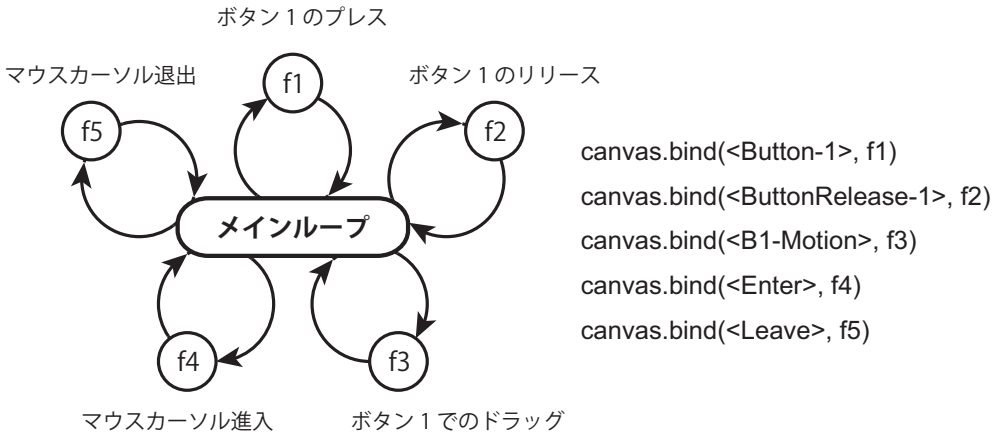

Figure 4.1: Tkinter におけるコールバック関数によるイベント処理

4.3 マウスイベントのプログラム

例１： マウスイベントによる背景色の変更 — background.py
　これは Canvas 内でのマウスイベントにあわせて背景色を変更するプログラムで，通常は図 4.2 左のように背景が緑色であるが，マウスボタンをプレスしている間だけ図右のように赤色に変化する．つまり，Canvas でイベントが発生すると，背景色を変更するプログラムになっている．pressed 関数ならびに released 関数において描画のために canvas を必要とするが，関数実行時に引数で渡されるのはイベントオブジェクト[1]だけなので，canvas を大域変数として main，pressed，released の 3 つの関数で共有している．

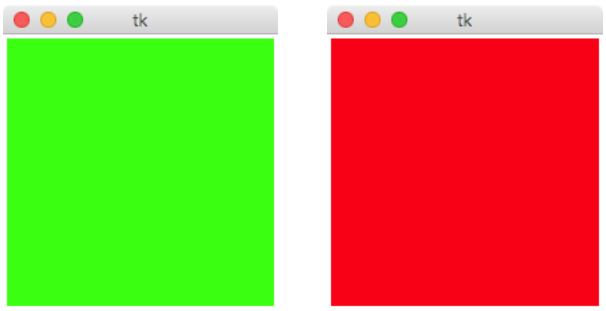

Figure 4.2: マウスイベントによる背景色の変更

```
from tkinter import *                    # tkinter モジュールの import
W, H = (200, 200)                        # canvas の幅と高さ
def pressed(event):                      # Button1 pressed コールバック関数
```

[1]実際にはイベントの発生した場所として canvas を調べることも可能である．

```
  global canvas                              # 大域変数 canvas
  canvas.create_rectangle((2, 2), (W+3, H+3), outline='', fill='#ff0000')
                                             # 背景の赤長方形での描画
def released(event):                         # Button1 released コールバック関数
  global canvas                              # 大域変数 canvas
  canvas.create_rectangle((2, 2), (W+3, H+3), outline='', fill='#00ff00')
                                             # 背景の緑長方形での描画
def main():                                  # main 関数
  global canvas                              # 大域変数 canvas
  root = Tk()                                # ルートフレームの作成
  canvas = Canvas(root, width = W, height = H, bg='#00ff00') # canvas の作成
  canvas.pack()                              # canvas の配置確定
  canvas.bind('<Button-1>', pressed)     # Button1 pressed コールバック関数
  canvas.bind('<ButtonRelease-1>', released) # Button1 released コールバック関数
  root.mainloop()                            # ルートフレームの実行ループ開始

if __name__ == '__main__':                   # 起動の確認（コマンドラインからの起動）
  main()                                     # main 関数の呼出
```

例２： カーソル位置の獲得と描画 — marker.py

このプログラムはボタンが押されるたびに，カーソル位置に正方形を描画する．図 4.3 のように押されたボタンによって正方形の色が変わる．イベント発生時のカーソル位置は，コールバック関数に渡された event オブジェクトの x フィールドと y フィールドで確かめられる．

```
from tkinter import *                        # tkinter モジュールの import

R = 5                                        # marker の半径（1 辺の長さの半分）

def pressed1(event):                         # Button1 pressed コールバック関数
  global canvas                              # 大域変数 canvas
  x, y = (event.x, event.y)                  # 現在の座標
  canvas.create_rectangle((x-R, y-R), (x+R+1, y+R+1),
                          outline='', fill='#ff0000') # 赤長方形（marker）の描画
def pressed2(event):                         # Button2 pressed コールバック関数
  global canvas                              # 大域変数 canvas
  x, y = (event.x, event.y)                  # 現在の座標
  canvas.create_rectangle((x-R, y-R), (x+R+1, y+R+1),
                          outline='', fill='#00ff00') # 緑長方形（marker）の描画
def pressed3(event):                         # Button3 pressed コールバック関数
  global canvas                              # 大域変数 canvas
  x, y = (event.x, event.y)                  # 現在の座標
  canvas.create_rectangle((x-R, y-R), (x+R+1, y+R+1),
                          outline='', fill='#0000ff') # 青長方形（marker）の描画
def main():                                  # main 関数
  global canvas                              # 大域変数 canvas
  W, H = (400, 400)                          # canvas の幅と高さ
  root = Tk()                                # ルートフレームの作成
  canvas = Canvas(root, width = W, height = H, bg='#ffffff') # canvas の作成
  canvas.pack()                              # canvas の配置確定
  canvas.bind('<Button-1>', pressed1)    # Button1 pressed コールバック関数
  canvas.bind('<Button-2>', pressed2)    # Button2 pressed コールバック関数
  canvas.bind('<Button-3>', pressed3)    # Button3 pressed コールバック関数
  #canvas.bind('<Shift-Button-1>', pressed3) # Shift+Btn1 pressed コールバック関数
  root.mainloop()                            # ルートフレームの実行ループ開始

if __name__ == '__main__':                   # 起動の確認（コマンドラインからの起動）
  main()                                     # main 関数の呼出
```

例３： マウスによる線分の描画 — rubberband.py

<Button-n>イベントだけではなく<Bn-Motion>イベントを用いることで，マウスをドラッグ (ボタンをプレスしながら移動) する間，マウスの動きに合わせた線分を描くことができる．このような線分は，マウスカーソルによって延びたり縮んだりするように見えることから，ラバーバンド (ゴム紐) と呼ばれる

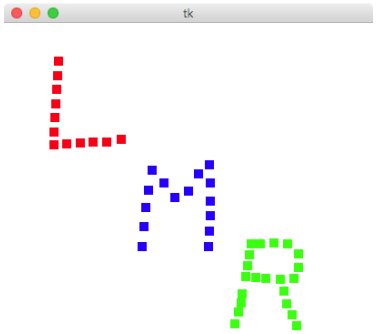

Figure 4.3: カーソル位置の獲得

ことが多い．最初にマウスボタンをプレスした位置が線分の始点となるため，pressed 関数と dragged 関数とで始点位置となる startX，startY を大域変数として共有している．

dragged 関数内で用いられている create_rectangle メソッドは，画面を背景色の長方形で塗りつぶすことで画面を消去するもので，線分が図 4.4 左のように毎回 1 本だけ描かれるようにしている．仮に，この create_rectangle メソッドをコメントアウトすると，図 4.4 右のように描画された線分がすべて画面上に残るため，マウスの移動に沿った複数の線分が表示されることになる．

```
from tkinter import *                    # tkinter モジュールの import

W, H = (400, 400)                        # canvas の幅と高さ

def pressed(event):                      # Button1 pressed コールバック関数
  global startX, startY                  # 大域変数 startX, startY
  startX, startY = (event.x, event.y)    # press 時の座標を記録

def dragged(event):                      # Button1 dragged コールバック関数
  global canvas, startX, startY          # 大域変数 startX, startY, canvas
  canvas.create_rectangle((2, 2), (W+3, H+3), outline='', fill='#ffffff')
                                         # 背景の白長方形での描画（クリア）
  x, y = (event.x, event.y)              # 現在の座標
  canvas.create_line((startX, startY), (x, y))
                                         # 線分の描画（press の座標から現在の座標まで）
def main():                              # main 関数
  global canvas                          # 大域変数 canvas
  root = Tk()                            # ルートフレームの作成
  canvas = Canvas(root, width = W, height = H, bg='#ffffff') # canvas の作成
  canvas.pack()                          # canvas の配置確定
  canvas.bind('<Button-1>', pressed)     # Button1 pressed コールバック関数
  canvas.bind('<B1-Motion>', dragged)    # Button1 dragged コールバック関数
  root.mainloop()                        # ルートフレームの実行ループ開始

if __name__ == '__main__':               # 起動の確認（コマンドラインからの起動）
  main()                                 # main 関数の呼出
```

例４： マウスを用いたお絵描き — draw.py

Rubberband では，マウスボタンがプレスされた位置とドラッグされたマウスカーソルの位置を結ぶように線分を描いたが，マウスカーソルの移動に従って，次々に短い線分を描画すれば，図 4.5 のようにマウスカーソルの軌跡が線として描かれる．

```
from tkinter import *                    # tkinter モジュールの import
```

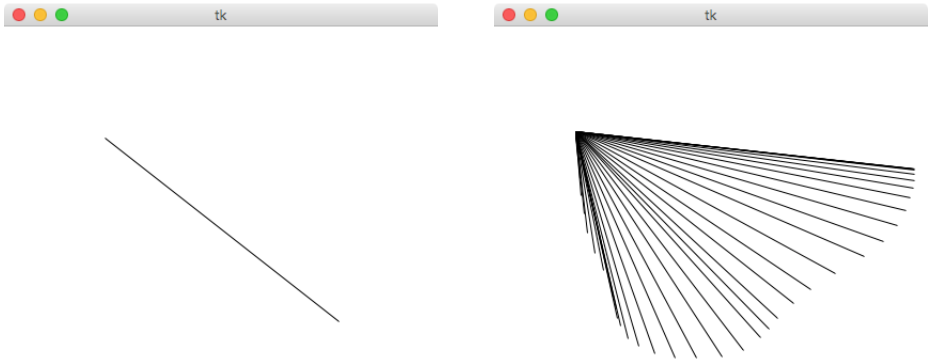

Figure 4.4: マウスの動きに伴う連続的な描画

```
def pressed(event):                          # Button1 pressed コールバック関数
  global oldX, oldY                          # 大域変数 oldX, oldY
  oldX, oldY = (event.x, event.y)            # press 時の座標を記録

def dragged(event):                          # Button1 dragged コールバック関数
  global canvas, oldX, oldY                  # 大域変数 oldX, oldY, canvas
  x, y = (event.x, event.y)                  # 現在の座標
  canvas.create_line((oldX, oldY), (x, y)) # 線分の描画（直前の座標と現在の座標）
  oldX, oldY = (x, y)                        # 座標の更新

def main():                                  # main 関数
  global canvas                              # 大域変数 canvas
  W, H = (400, 400)                          # canvas の幅と高さ
  root = Tk()                                # ルートフレームの作成
  canvas = Canvas(root, width = W, height = H, bg='#ffffff') # canvas の作成
  canvas.pack()                              # canvas の配置確定
  canvas.bind('<Button-1>', pressed)         # Button1 pressed コールバック関数
  canvas.bind('<B1-Motion>', dragged)        # Button1 dragged コールバック関数
  root.mainloop()                            # ルートフレームの実行ループ開始

if __name__ == '__main__':                   # 起動の確認（コマンドラインからの起動）
  main()                                     # main 関数の呼出
```

章末課題

マウスイベントの確認

例１「マウスイベントによる背景色の変更」を参考に，次のような動きをするプログラムを作成して，その挙動を調べよ．Python の実行環境によっては，挙動が変わる可能性があるので，実行した環境について言及すること．

1. 初期状態では背景色は白．
2. 背景色を，マウスボタンのプレスで赤，リリースで緑，マウスカーソルの進入で青，
   マウスカーソルの退出で黄色へと，それぞれ変更する．

Figure 4.5: マウスによるお絵描き

マウスによる円の描画

例 3「マウスによる線分の描画」を参考に，次のような動きをする Canvas プログラムを作成せよ．

1. マウスボタンをプレスしてドラッグすると円を描く．
2. ボタンをプレスした位置が円の中心となる．
3. マウスをドラッグすると，円はカーソル位置を通過するように拡大縮小する．
4. 円の描画には，次のように create_oval メソッドを用いるとよい（引数は create_rectangle メソッドと同じで，長方形内部に内接する楕円を描く）．
   canvas.create_oval((左上 x 座標，左上 y 座標)，(右下 x 座標，右下 y 座標))

参考：プログラムの改良

以下のプログラムを解読して，プログラムを改良せよ．

```
from tkinter import *                    # tkinter モジュールの import
import random                            # random モジュールの import
import time                              # time モジュールの import

W, H  = (400, 400)                       # canvas の幅と高さ
R     = 5                                # 正方形（marker）の半径（1 辺の半分）
TIMES = 10                               # 試行回数

def display():                           # 正方形（marker）の描画関数
  global canvas, error, x, y             # 大域変数 canvas, error, x, y
  x = random.randint(2+R, W+2-R)         # 正方形（marker）の x 座標
  y = random.randint(2+R, H+2-R)         # 正方形（marker）の y 座標
  canvas.create_rectangle((2, 2), (W+3, H+3), outline='', fill='#ffffff')
                                         # 背景の白長方形での描画（クリア）
  if error:                              # ミスした場合 - 赤で描画
    canvas.create_rectangle((x-R, y-R), (x+R, y+R), outline='', fill='#ff0000')
  else:                                  # ヒットした場合 - 黒で描画
    canvas.create_rectangle((x-R, y-R), (x+R, y+R), outline='', fill='#000000')

def pressed1(event):                     # Button1 pressed コールバック関数
  global x, y, error, count, ttime, fastest # 大域変数 x, y, error など
  if count > 0:                          # カウント（残り回数）が正 = 実行中
    xc, yc = (event.x, event.y)          # press 時の座標
    if x - R <= xc <= x + R and y - R <= yc <= y + R: # 正方形（marker）との位置
      error = False                      # 正方形（marker）の外部 - ミス
      count = count - 1                  # カウント（残り回数）の減少
    else:
```

```
      error = True                          # 正方形（marker）の内部 - ヒット
      count = count + 1                     # カウント（残り回数）の増加
    if count > 0:                           # カウント（残り回数）が正 = 継続
      print(count, 'more!!')                # カウント（残り回数）の表示
      display()                             # 正方形（marker）の描画
    else:                                   # カウント（残り回数）が 0 = 終了
      ttime = time.time() - ttime           # 作業時間の測定
      if fastest < 0 or ttime < fastest: # 最速時間の場合
        fastest = ttime                     # 最速時間の更新
      print('Finished in', ttime, 'secs. Fastest time:', fastest, 'secs.')
                                            # 最速時間の表示
  else:                                     # カウント（残り回数）が 0 = 実行前
    if fastest < 0:                         # (最速時間の) 記録なしの場合
      print('Click right button to start.') # 開始方法のメッセージ表示
    else:                                   # (最速時間の) 記録ありの場合
      print('Click right button to start. Fastest time:', fastest, 'secs.')
                                            # 開始方法のメッセージと最速時間記録の表示
def released2(event):                       # Button2 released コールバック関数
  global error, count, ttime                # 大域変数 error, count, ttime
  error = False                             # ヒット／ミスの初期化 - ヒット
  count = TIMES                             # カウント（残り回数）の初期化
  print('Start clicking ...', count, 'more!!') # 開始とカウント（残り回数）の表示
  ttime = time.time()                       # 開始時刻の記録
  display()                                 # 正方形（marker）の描画

def main():                                 # main 関数
  global canvas, count, fastest             # 大域変数 canvas, count, fastest
  count, fastest = (0, -1)                  # カウント（残り回数）と最速時間の初期化
  root = Tk()                               # ルートフレームの作成
  canvas = Canvas(root, width = W, height = H, bg='#ffffff') # canvas の作成
  canvas.pack()                             # canvas の配置確定
  canvas.bind('<Button-1>', pressed1)    # Button1 pressed コールバック関数
  canvas.bind('<ButtonRelease-2>', released2) # Button2 released コールバック関数
  root.mainloop()                           # ルートフレームの実行ループ開始

if __name__ == '__main__':                  # 起動の確認（コマンドラインからの起動）
  main()                                    # main 関数の呼出
```