はじめに

ゲームのコンセプトは、「簡単なミニゲームをテンポよくこなしていくゲーム」である。任天堂の「メイドインワリオ」というゲームを参考にした。具体的には、「矢印キーを押すだけのタスク」や「クリックするだけのタスク」などの非常に操作性のシンプルなゲームを作ることで、「老若男女問わず誰でも楽しめるゲーム」を目指した。

ゲームの概要

今回作成したゲームは以下のようなフローチャートに従って処理を行う。

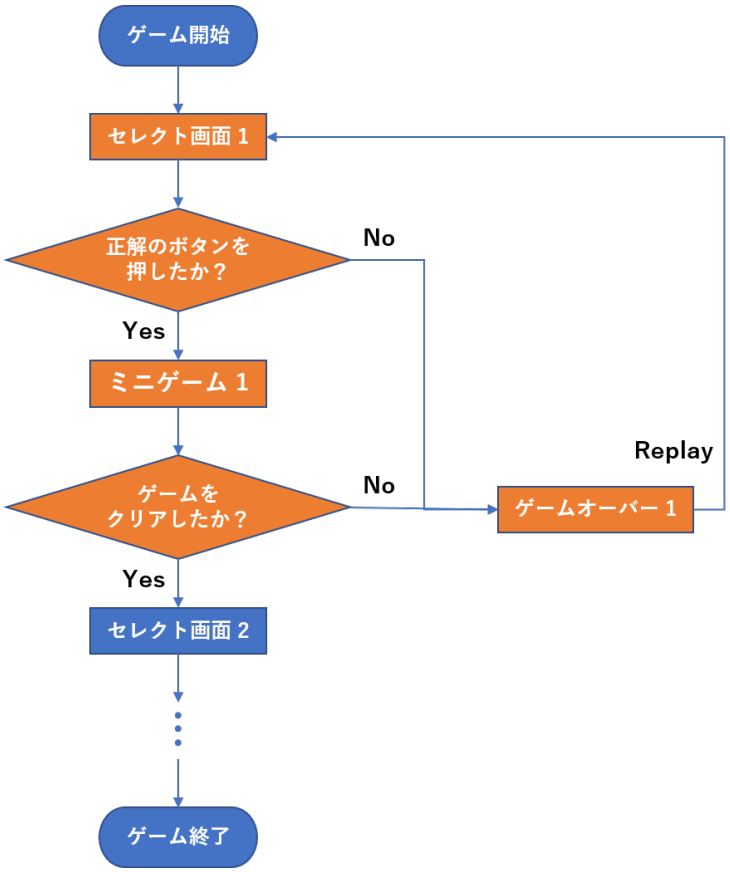

図 1：ゲームのフローチャート

全ての処理を記述するとフローチャートが長くなるため、一部を抜粋した。基本的に「オレンジ色の処理」を繰り返すだけである。まず、セレクト画面にある 2 つのボタンのうち 1 つを選ぶ。失敗のボタンであればゲームオーバー画面に進む。正解のボタンであれば、ミニゲームに進める。ミニゲームを行いクリアすれば次のステージへ進み、失敗すればゲームオーバー画面に進む。これが基本の処理である。

ミニゲーム

本レポートで解説するゲームは、全体の処理をコントロールする「final_main.py」と 7 つのミニゲーム「final_maze.py」、「final_block.py」、「final_bar.py」、「final_chase.py」、「final_crash.py」、「final_shooting.py」、「final_boss.py」から構成される。プログラムが非常に長いので、「変数の宣言部分」、「画像や音声の読み込み」や「マルチメディア演習で解説されたプログラムの技法」などの解説は省略する。

本レポートでは、作者が工夫した点に焦点を当てて解説を行う。ソースコードに細かい解説が書いてあるので、ソースコードも見てほしい。また、ゲームは実際にプレイした方が速く理解できると思われる。※ソースコードは背景が黒くなっている。

final_main.py

```
page = 0
```

まず、page という変数を使って、ステージの遷移を行う。例えば、page=0 ならタイトル画面に遷移し、page=1 ならセレクト画面に遷移する。

```
def button_to_jump(btn, newpage):
    global page, pushFlag, shake #グローバル変数にして全体に影響

    mdown = pg.mouse.get_pressed() #マウスのボタンの種類
    (mx, my) = pg.mouse.get_pos() #マウスの位置
    if mdown[0]:
        #ボタンをクリック かつ pushplag が False のとき
        if btn.collidepoint(mx, my) and pushFlag == False:
            pg.mixer.Sound("sounds/kyuin.mp3").play() #ボタンを押した音
            page = newpage #ページ(画面)を遷移
            pushFlag = True #クリックした印
            shake = 10 #(処理の関係上)画面が遷移したらリセット
        else:
            pushFlag = False #(処理の関係上)False にする
```

次に、遷移は button_to_jump という関数を使用する。この関数は、画面上のボタンが押された際に page の値を更新する機能を持つ。ボタンは、「Replay ボタン（タイトルに戻る)」、「戻るボタン（1 つ前のセレクト画面に戻る)」などがある。

```
def maze2(): #迷路ゲーム
    global page
    state = final_maze.maze() #クリアしたらTrueを返す、失敗したらFalseを返す関数
    if state == False: #失敗→ゲームオーバー
        page = 3 #ゲームオーバー
    elif state == True: #成功→次のステージへ
        page = 4 #次のステージへ
```

final_main.py には、各ステージの処理が関数として定義されている。関数名は「ステージ名＋page の値」になっている。例えば、ミニゲームの 1 つを定義している関数は「maze2」となっており、ステージ名が「maze」、page の値が「2」であることがわかる。ミニゲームは、外部のファイルから関数を読み込んで（import して）実行され、クリアした場合は True が、失敗した場合は False が返される。

関数の主な種類は、「セレクト画面」、「ミニゲーム」、「ゲームオーバー画面」の 3 つである。セレクト画面とゲームオーバー画面では、主に背景画像を張り付け、ボタンを置く処理をしている。

```
while True:
    if page == 0:
        title0()

    elif page == 1:
        select1()
```

上記のように、ステージの遷移は、if 文で page の値を条件分岐して実行される。if 文は全部で 23 個ある。すなわち。23 のステージが定義されていることがわかる。

ゲームで共通の処理（カウントダウン）

各ミニゲームでは、制限時間が設定されている。制限時間（カウントダウン）の表示は以下のように行う。

```
    second = 12 #12秒以内
    total = 0 #時間計測で使用
```

まず、上記のように変数を定義する。second は残り時間を表す変数、total は時間計測で使用する変数である。

```
while True:
        if second > 0: #残り時間が0秒より大きかったら
            start = time.perf_counter() #計測スタート
```

次に、ゲーム処理のループの最初に time ライブラリの time.pref_counter 関数を使って時間の計測を始める。start に while ループの先頭行が実行された時の時刻が入る。

```
        if second > 0: #残り時間が 0 秒より大きかったら
            end = time.perf_counter() #計測終了
            p_time = end - start #1 つの処理にかかった時間を求める
            total += p_time #1 つの処理にかかった時間を累計

            if total >= 1: #累計時間が 1 秒を超えたら
                second -= 1 #残り時間を 1 秒減らす
                total = 0 #累計時間をリセット
```

画面描画などゲーム処理のループが 1 回終了したら、上記のように時間計測を終了する。end に処理が終わったときの時刻が入るので、「end-start」を計算することで、ループ 1 回分の処理時間（p_time）を得ることができる。この処理時間を total という変数に累積していく。ループが繰り返されると、total の値が 1（秒）を超えるので、その時、残り時間を表す second から 1 を引く。この処理によって、カウントダウンを実現することができる。

この方法では、正確にカウントダウンを実行することができないが、誤差は 1/100 秒程度であり、（人間は認識ができないので）ゲームの中で使う分には問題は起こらない。

```
        font = pg.font.Font("ipaexg.ttf", 40)
        text = font.render("のこり"+str(second)+"秒", True,
pg.Color("WHITE"))
        screen.blit(text, (25, 20))
```

また、上記のようにカウントダウンを画面に出力する。まず、設定したフォントを font に代入して、表示する文字を text に代入する。その後、screen.blit メソッドで座標を設定して画面に text を張り付ける。

final_maze.py

このプログラムは、迷路ゲームの処理を記述している。

```
def make_maze():
```

まず、迷路がランダムで作成されるように、make_maze という関数を定義している。具体的には、配列に通路（0）と壁（1）に相当する値を代入する処理を行っている。

```
      for y in range(2, MAZE_H-2, 2): #3, 5 行目
          for x in range(2, MAZE_W-2, 2): #カード（アイテム）を置く 9 列目にはラ
ンダムに壁を作らない（3, 5, 7 列目）
              d = random.randint(0, 3) #0 から 3 までの乱数
              if x > 2: #3 だったら
                  d = random.randint(0, 2)
```

```
                maze[y+YP[d]][x+XP[d]] = 1
```

ただし、迷路の一番右の通路（配列の 9 列目）には、アイテム（カード）が置かれるので、その列には壁ができないように、ランダム壁を作る範囲を制限している。

```
def draw(screen):
```

次に、draw_screen 関数で画面に迷路やプレイヤーなどを出力する。

```
    cardFlag = False #カードをゲットしたら True
```

```
        #カード(アイテム)を拾ったかの判定
        if cardRect.colliderect(myrect): #プレイヤーがアイテムに触れたら
            pg.mixer.Sound("sounds/card.mp3").play()
            cardFlag = True #カードをゲットした印
            #カードの座標を画面外にしてカードを消す
            cardRect.x = 800
            cardRect.y = 600
```

上記がアイテム（カード）をゲットした場合の処理である。カードをゲットした印である cardFlag を True にしていることがわかる。

```
    def move_player():
        global pl_x, pl_y, goalFlag
        key = pg.key.get_pressed() #押されたキーを取得
        if key[pg.K_UP] == 1: #上方向のキーなら
            if maze[pl_y-1][pl_x] != 1: #壁(1)に当たらなければ
                pl_y = pl_y - 1 #上に動く(y 方向に-1 する)
                myrect.y -= H #上に1ブロック分動く(画面上の座標の設定)
                pg.mixer.Sound("sounds/walk.wav").play() #足音
```

さらに、プレイヤーの動きを設定する move_player 関数定義している。この関数では、プレイヤーの進む方向に壁が無ければ、プレイヤーを画面上で移動させる処理を行う。変数 pl_x と pl_y で迷路（配列）上でのプレイヤーの位置を管理している。上記は、上方向に進む際の処理である。

```
    goalFlag = False #ゴールしたら True
```

```
        #クリアしたかどうかの判定
        if (goalFlag == True) and (cardFlag == True): #アイテムをゲットしてゴールしたら
```

```
            pg.mixer.Sound("sounds/shine.mp3").play()
            return True #True(成功)を返す
        elif second <= 0: #制限時間オーバーのとき
            pg.mixer.Sound("sounds/down.wav").play()
            return False #False(失敗)を返す
```

上記のように、アイテムを取ったら True になる「cardFlag」とゴールしたら True になる「goalFlag」を定義し、if 文などを使って、状況に応じた処理をしている。

final_block.py

このプログラムは、ブロック崩しをベースにしたゲームの処理を記述している。

```
state = 1 #初期の状態
```

まず、state という変数を用いて現在の状態を表す。

```
        if state == 1 and (second > 0): #通常
            gamestage()
        elif state == 2 or (second <= 0): #失敗
            pg.mixer.Sound("sounds/down.wav").play()
            return False
        elif state == 3: #クリア
            pg.mixer.Sound("sounds/shine.mp3").play()
            return True
```

例えば、state=1 ならゲーム続行、失敗した時は state=2、クリアしている時は state=3 となる。

```
def gamestage():
```

次に、ブロック崩しのステージを作成する gamestage 関数が定義されている。ボールが画面の縁にぶつかったときの処理やブロックとぶつかったときの処理が colliderect 関数などを用いて記述されている。

```
score = 0 #スコア
```

```
        if score == 10: #全てのブロックに当てたらボスが出る
            screen.blit(bossimg, bossrect)
            if bossrect.x <= 0 or bossrect.x >= 800-80:
                bv = -bv #逆方向に動く
            bossrect.x += bv
            if bossrect.colliderect(ballrect): #ボスとボールが当たったら
```

```
                state = 3 #状態を 3 に更新（クリア）
```

また、ブロックを消すごとに score が 1 ずつ増加していく。ブロックの総数は 10 個なので、score が 10 になったら、ボス（「追試」という画像）を出現させるようにする。ボスは x 軸方向に速度 bv で動き、ボスにボールを当てれば、クリア（state=3）となる。

final_bar.py

このプログラムは、「制限時間内でボールを落としてはいけないゲーム」をベースにしたミニゲームの処理を記述している。

```
state = 1 #初期の状態
```

他のプログラムと同様に、state で現在のゲームの状態を管理している。

```
        if (state == 1) and (second > 0): #通常
            gamestage()
        elif state == 2: #失敗
            pg.mixer.Sound("sounds/down.wav").play()
            return False
        elif second <= 0: #クリア
            pg.mixer.Sound("sounds/shine.mp3").play()
            return True
```

例えば、state=1 ならゲーム続行、state=2 ならクリア失敗を表す。

```
def gamestage():
```

次に、このゲームのステージを作成する gamestage 関数が定義されている。
この関数内で、ボールやボールを跳ね返すバーの設定などが行われている。

```
        #ボール 2 つ目
        if second <= 14: #14 秒以下になったらボール 2 つ目を出す
            if ballrect2.y < 0: #上の壁に当たったら
                vy2 = -vy2 #逆方向に動く
            if ballrect2.x < 0 or ballrect2.x > 800 - 70: #左右の壁に当たっ
たら

                vx2 = -vx2 #逆方向に動く
            if barrect.colliderect(ballrect2): #ボールとバーがぶつかったら
                vx2 = random.randint(-15, 15)
                vy2 = random.randint(-15, -10) #上方向の速度はランダム
                pg.mixer.Sound("sounds/doko.wav").play()
            if ballrect2.y > 600: #画面の下に落ちたら
```

```
                state = 2 #状態を 2 に更新(失敗)
            ballrect2.x += vx2
            ballrect2.y += vy2
            screen.blit(ballimg, ballrect2)
```

上記のように、残り時間（second）が一定時間（14 秒）以下になったら新たなボール（「中間」と「期末」という画像）を画面に出す処理を行う。14 秒と 7 秒で新たなボールが落ちてきて、合計でボールは 3 つになる。

```
itemFlag = False #上から落ちてくるアイテムをゲットしたら True
```

```
        #アイテムの処理
        if second <= 15: #残り時間が 15 秒になったらアイテムを落とす
            if itemrect.y <= 600: #画面の下に到達するまで
                    itemrect.y += 4
                    screen.blit(itemimg, itemrect)
            if itemrect.y > 600: #画面の下に来たら
                    itemrect.y = 700 #アイテムを消す
            if itemrect.colliderect(barrect): #アイテムがバーと当たったら
                pg.mixer.Sound("sounds/magic.wav").play()
                itemFlag = True #アイテムをゲットした印
```

さらに、上記のように、残り時間が 15 秒以下になったらアイテムを画面に描画する処理を行う。落ちてくるアイテムをプレイヤーがゲットしたら itemFlag が True になる。

```
        if itemFlag == False: #アイテムをゲットする前
            barrect.x = mx - 50 #マウスの x 座標の-50 にバーを作る(バーの x 座標の中心)
            pg.draw.rect(screen, pg.Color("CYAN"), barrect) #バーを出力
        else: #アイテムをゲットした後
            itemrect.y = 700 #アイテムを消す
            barrect = pg.Rect(400, 500, 300, 20) #バーを大きくする
            barrect.x = mx - 150 #マウスをバーの x 座標の中心に
            pg.draw.rect(screen, pg.Color("GOLD"), barrect) #バーを出力
```

itemFlag が True になったとき（else の部分）は、バーを大きくして色をゴールドに変更する。これによって、（アイテムによって）バーがパワーアップしたことを認識できる。

final_chase.py

このプログラムは、「敵が追いかけてきて制限時間以内にゴールへ向かうゲーム」をベースにしたゲームの処理を記述している。

```
state = 1 #初期の状態
```

まず、state という変数を用いて現在の状態を表す。

```
        if state == 1 and (second > 0): #通常
            gamestage()
        elif (state == 2) or (second <= 0): #失敗
            pg.mixer.Sound("sounds/down.wav").play()
            return False
        elif state == 3: #成功
            pg.mixer.Sound("sounds/shine.mp3").play()
            return True
```

例えば、state=1 ならゲーム続行、state=2 なら失敗、state=3 ならクリアを表す。

```
def gamestage():
```

次に、このゲームのステージを作成する gamestage 関数が定義されている。「コンピュータのキーを押してプレイヤーを画面上で移動させる処理」や「オブジェクトとの衝突判定」などを行っている。衝突判定には、collidelist 関数や colliderect 関数を利用している。

```
        #偶数個目のトラップを左右に動かす
        for trap in traps[0::2]: #0 から 2 ずつインデックスを増加
            if second % 2 == 0: #残り時間が偶数の時
                trap.x += 1 #左方向に+1
            else:
                trap.x -= 2 #右方向に+2
        #奇数個目のトラップを上下に動かす
        for trap in traps[1::2]: #1 から 2 ずつインデックスを増加
            if second % 2 == 0: #残り時間が偶数の時
                trap.y += 2 #下方向に+2
            else:
                trap.y -= 2 #上方向に+2
```

また、画面上でトラップを動かす処理を行った。トラップの土台（trap）は「配列」に格納されているので、偶数番目の土台を上下に、奇数番目の土台を左右に動かすように設定した。さらに、左右に動く土台は左から右に少しずつ移動するように速度を調整した。

具体的には、残り時間が偶数の時、インデックスが偶数番目のトラップは右方向に、インデックスが奇数番目のトラップは上方向に移動する。残り時間が奇数の時は、逆方向の動き

をするように設定した。

final_crash.py

このプログラムは、「制限時間以内にターゲットに触れるゲーム」をベースにしたミニゲームの処理を記述している。

```
state = 1 #初期の状態を示す
```

他のプログラムと同様に、state で現在のゲームの状態を管理している。

```
        if state == 1 or (second > 0): #通常
            gamestage()
        elif state == 2 and (second <= 0): #クリア
            pg.mixer.Sound("sounds/shine.mp3").play()
            return True
        elif state == 3: #失敗
            pg.mixer.Sound("sounds/down.wav").play()
            return False
```

例えば、state=1 ならゲーム続行、state=2 なら失敗、state=3 ならクリアを表す。

```
def gamestage():
```

次に、このゲームのステージを作成する gamestage 関数が定義されている。

```
#ターゲットが壁に当たったら左右に動くように処理
    if crashflag == False: #ターゲットに触れていないとき
        if virusrect.x < 5 or virusrect.x > 800-70: #左右の壁に当たったら
            vx = -vx #逆方向に動く
        virusrect.x = virusrect.x + vx

        #ターゲットが画面内で上下に動くように処理
        if (virusrect.x>= 0 and virusrect.x < 200) or (virusrect.x>= 400
and virusrect.x < 600):
            virusrect.y = virusrect.y + 11
        elif (virusrect.x>= 200 and virusrect.x < 400) or (virusrect.x>=
600 and virusrect.x < 800-70):
            virusrect.y = virusrect.y - 9
        screen.blit(virus_img, virusrect) #virusrect の上に画像を張り付ける

        #手の位置の設定
        (mx, my) = pg.mouse.get_pos()
```

```
        #手の画像の中心とカーソルの位置が同期するようにする
        handrect.x = mx - 65
        handrect.y = my - 60
        screen.blit(handimg, handrect)
```

この gamestage 関数には、「ターゲット（ウイルス）が上下左右に動く処理」や「手（猫の手）をカーソルの動きと同期させる処理」などが記述されている。具体的には、ターゲットの土台（virusrect）と手の土台（handrect）の座標を細かく設定している。

```
crashflag = False #ターゲットに触れたら True
```

また、ターゲットに触れたかどうかを判断する crashflag を定義した。

```
    if crashflag == True: #ターゲットに触れたら
        lost_img = pg.transform.rotate(virus_img, 180) #ターゲットがひっくり返
る
        screen.blit(lost_img, virusrect) #背景画面を変える
        #叩いたら手が震えるようにする
        (mx, my) = pg.mouse.get_pos()
        handrect.x = mx - 65 + random.randint(-3, 3) #左右に小刻みに震える
        handrect.y = my - 60 + random.randint(-3, 3) #上下に小刻みに震える
        screen.blit(handimg, handrect)
```

ターゲットに触れたとき（crashflag が True のとき）、これまでのゲーム画面の出力をやめ、transform.rotate 関数でターゲットを 180°ひっくり返して「倒された感じ」を出し、「よくできました」という判子の画像を画面に出力する。

さらに、ターゲットを叩いた手を小刻みに震えさせて「痛そうな感じ」を出す。具体的には、手の x 座標と y 座標をランダムに-3 から 3 の間で動かして、画面が更新される度に手が震えるようにする。

final_shooting.py

このプログラムは、「シューティングゲーム」をベースにしたゲームの処理を記述している。

```
state = 1 #初期の状態
```

他のプログラムと同様に、state で現在のゲームの状態を管理している。

```
score = 0 #スコア
```

```
        if (state == 1) and (second > 0): #通常
            gamestage()
        elif (state == 2) or (score < 10000): #失敗
            pg.mixer.Sound("sounds/down.wav").play()
            return False
        elif (second <= 0) and (score >= 10000): #クリア
            pg.mixer.Sound("sounds/shine.mp3").play()
            time.sleep(0.5)
            return True
```

例えば、state=1 ならゲーム続行、state=2 なら失敗、state=3 ならクリアを表す。また、「敵を倒したとき」や「当たらずに動き続けたとき」に得点（score）が加算される。この score を制限時間内が 0 秒になる前に 10000 点以上にすれば、クリアとなる。

```
def gamestage():
```

次に、このゲームのステージを作成する gamestage 関数が定義されている。画面がスクロールしているように見えるように、桜の画像が動いて出力されたり、敵や弾の設定が記述されたりしている。

```
specialFlag = False #1 度特別な弾を打ったら True
```

```
#特別な弾(1 回限り)
if specialFlag == False: #specialFlag が False のとき使える
    if mdown[2] and specialrect.y < 0: #右クリック、画面の下に準備してあったら
        specialrect.x = myrect.x + 70/2 - 175/2 #プレイヤーの中心から弾が出る
ように
        specialrect.y = myrect.y - 120
        pg.mixer.Sound("sounds/special.wav").play()
    if specialrect.y >= -175: #弾が画面に残っていたら
        specialrect.y -= 7 #弾を上へ進める
```

```
        specialrect.x = myrect.x + 50/2 - 175/2 #カーソルの動きと連動する
        if specialrect.y < -175:
            specialFlag = True
        screen.blit(specialimg, specialrect)
```

通常の弾に加え「特別な弾」の処理を用意した。この弾は一度しか使えないようにする。具体的には、一度でも弾を使用したら specialFlag を True にして、if 文で弾を使えなくする。また、この特別な弾は敵と当たっても消えずに、画面から出るまで出力され続ける。さらに、カーソルの動きと特別な弾の座標を一致させることで、弾を自由に動かせるようにしている。

```
                if report.colliderect(penrect):
                    report.y = -100 #画面外へ消える
                    report.x = random.randint(0+25, 800-25)
                    penrect.y = -100 #画面外へ消える
                    pg.mixer.Sound("sounds/piko.wav").play()
                    score += 100 #100 点入る
```

```
score = score + 10 #当たらずに動いていたら+10
```

弾が敵と当たったら、score に 100 点入るようにする。このとき、colliderect 関数を使って衝突判定を行う。また、score は画面が更新される度に 10 点入るようにする。

final_boss.py

このプログラムは、「攻撃をよけながらボスに攻撃を当てるボス戦」をベースにしたミニゲームの処理を記述している。

```
state = 1 #初期の状態
```

他のプログラムと同様に、state で現在のゲームの状態を管理している。

```
        if (state == 1) and (second > 0): #通通常
            gamestage()
        elif (clearFlag == True) and (second <= 0): #クリア
            pg.mixer.Sound("sounds/defeatboss.mp3").play()
            time.sleep(0.5)
            return True
        elif (state == 2) or (second <= 0): #失敗
            pg.mixer.Sound("sounds/down.wav").play()
            return False
```

例えば、state=1 ならゲーム続行、state=2 なら失敗を表す。

```
def gamestage():
```

次に、このゲームのステージを作成する gamestage 関数が定義されている。この関数では、ボスの動き方、ボスの攻撃方法、プレイヤーが放つ弾などの処理が記述されている。

```
clearFlag = False #クリアしたら True
```

クリアしたかどうかを判定する clearFlag を定義した。

```
        if clearFlag == True: #クリア後の画面表示
            screen.fill(pg.Color("WHITE"))
            screen.blit(naitei, (0,0))
            font = pg.font.Font("ipaexg.ttf", 40)
            text = font.render("おめでとう！", True, pg.Color("RED"))
            screen.blit(text, (480, 480))
```

clearFlag が Ture のとき（クリアしたとき）、「おめでとう」という文字と共に背景に祝福の画像が出力されるようになっている。このとき、操作は何もできないようになっている（後述の「clearFlag が False のとき（クリアしていないとき）」で様々なゲーム画面の操作を定義している）。

```
if clearFlag == False:
```

以下では、clearFlag が False のとき（クリアしていないとき）の処理を解説する。

```
bulletFlag = False #1 回弾を打ったら True
```

弾を一発打ったかどうかを示す bulletFlag を定義している。

```
            if bulletFlag == False:
                if mdown[2] and bulletrect.y < 0: #左クリックかつ画面の上から出たら
                    bulletrect.x = myrect.x + 120/2 - 64/2 #プレイヤーの中心から弾が出るように
                    bulletrect.y = myrect.y #プレイヤーの y 座標から弾が出るように
                    pg.mixer.Sound("sounds/special.wav").play()
                if bulletrect.y >= -120/2: #弾が画面に残っていたら
                    bulletrect.y -= 28 #弾を上へ進める
                    screen.blit(bulletimg, bulletrect)
                    if bulletrect.y <= -120/2:
                        bulletFlag = True
```

上記が弾の処理である。今回のゲームでは、「弾は一発のみ」で、ボスに当てられない場合は制限時間が尽きるのを待つだけになる。これを実現するため、弾を一度でも使用し

たら bulletFlag を True にして、if 文で弾を使えなくする。また、弾の動き方や描画は、前述のシューティングゲーム（final_shooting.py）と同様である。

```
            #ボスの動きの処理
            if bossrect.x == 340: #画面の中心でボスの速度をランダムに変える
                if bv < 0:
                    bv = random.randrange(-40, -10, 10)
                elif bv > 0:
                    bv = random.randrange(10, 40, 10)
            if bossrect.x <= -20 or bossrect.x >= 800-100: #左右の壁の部分
に着いたら
                bv = -bv #逆方向に動く
            bossrect.x += bv
            screen.blit(bossimg, bossrect)
            if bossrect.colliderect(bulletrect): #ボスと弾が当たったら
                pg.mixer.Sound("sounds/up.wav").play()
                clearFlag = True #クリアした印
```

上記の部分では、ボスの動き方の処理が記述されている。ボスは、左右にのみ動き、速度は bv で管理される。

具体的には、ボスの x 座標が画面の中心くらい（340）になったら、ランダムでボスの速度を変化させている。乱数は 10 の倍数にしないと、ボスの x 座標が 340 にならない可能性があるので、乱数の値は 10, 20, 30, 40 のどれかである。また、「壁に当たったら逆方向に動く処理」、「弾と当たったらクリアした印として clearFlag を Ture にする処理」などが記述されている。

```
enemyFlag = False #enemy が画面にいたら True
```

「ボスから発射される敵が画面内にいるかどうか」を表す enemyFlag を定義している。

```
#ボスから発射される敵の処理
            if enemyFlag == False: #敵が画面にいないとき
                pg.mixer.Sound("sounds/apper.wav").play()
                if bv > 0: #ボスが右方向に動くとき
                    enemyrect.x = bossrect.x + 100/2 #ボスから出す
                elif bv < 0: #ボスが左方向に動くとき
                    enemyrect.x = bossrect.x - 100/2
```

上記がボスから発射される敵の処理である。enemyFlag が False のとき（画面に敵がいないとき）に、この処理は行われる。これによって、ボスは 1 体ずつ敵を発射することにな

る（難易度調整のため 1 体ずつ発射するようにした）。

また、ボスの動きが非常に速いため、敵を発射する位置を少しずらしている。具体的には、ボスの速度が右向きの時は、敵（enemyrect）の x 座標を 50 だけ右にずらし、ボスの速度が左向きの時は、敵（enemyrect）の x 座標を 50 だけ左にずらしている。

```
            if enemyrect.y <= 600: #画面の下に到達するまで
                enemyFlag = True
                enemyrect.y += 25
                if enemyrect.y >= bossrect.y + 100/2: #ボスの y 座標より 50 下
へ来たら
                    screen.blit(enemyimg, enemyrect) #画面に出力
            if enemyrect.y > 600: #画面の下に来たら
                enemyFlag = False
                enemyrect.y = 50 #初期化（ボスの y 座標へ）
            if enemyrect.colliderect(myrect): #enemy がプレイヤーと当たったら
                state = 2 #状態を 2 に更新（失敗）
            if enemyrect.colliderect(bulletrect): #弾と enemyimg が当たったら
                enemyFlag = False
                enemyrect.y = 50 ##初期化（ボスの y 座標へ）
                pg.mixer.Sound("sounds/piko.wav").play()
```

上記がボスから敵の処理の続きである。ボスのいる位置から、垂直に落下することがわかる。基本的に前述のシューティングゲーム（final_shooting.py）と同様の処理を行っている。ただし、画面に出力する前、敵はボスの y 座標（50）の位置に待機させ、「ボスの y 座標+50」より下まで進んだら画面に出力するようにする。したがって、敵の座標が 50 の時（enemyrect.y=50）は、画面に出力されない。

final_main.py の go_to_boss19 関数について

煩雑になるため、この章で final_main.py の bo_to_boss19 関数について解説する。

```
shake = 10 #画面を揺らす回数（ボス戦の前に使用）
tick = 60 #画面の更新回数（ボス戦の前に使用）
```

```
def go_to_boss19(): #ボスへの挑戦前の画面
    global shake, tick
    screen = pg.display.set_mode((800, 600))
    if shake == 10:
        pg.mixer.Sound("sounds/warning.mp3").play()
```

```
    if shake > 0:
        tick = 5 #画面の更新回数を 60 から 5 に減らして画面が揺れる演出をする準備を
する
        shake = shake - 1 #揺らす回数を 0 まで減らす
        bx = random.randint(-20, 20) #左右に揺らす
        by = random.randint(-10, 10) #上下に揺らす
    elif shake == 0: #画面が揺れないようにする
        bx = 0
        by = 0
        tick = 60 #画面の更新回数を 60 に戻す
    screen.blit(img6, [0+bx, 0+by]) #画面が揺れる
    btn1 = screen.blit(boss_img, (575+bx, 526+by)) #ボス戦へのボタンも揺らす
    btn2 = screen.blit(again_img, (30+bx, 515+by)) #1つ戻るボタンも揺らす
    btn3 = screen.blit(replay_img, (630+bx, 5+by)) #リプレイボタンも揺らす
    button_to_jump(btn1, 20) #リプレイボタン
    button_to_jump(btn2, 16) #1 つ前のページに戻る
    button_to_jump(btn3, 0) #最初に戻る
```

基本的に背景画像（img6）とボタンを画面に出力する処理が記述されている。ただし、この関数では「画面を揺らす処理」が実行される。

ここで、「画面を揺らす残りの回数」は shake、「フレームレート」は tick で管理される。まず、shake が 0 になるまで、フレームレートを 5 に落とす（人間の目が反応できるようにフレームレートを落としている）。さらに、bx と by に乱数を代入する。その後、背景やボタンの x 座標と y 座標にそれぞれ bx と by を足すことで、更新されるたびに画面が揺れることになる。その他の処理は、セレクト画面やゲームオーバー画面と同様である。

おわりに

ゲーム作りの大変さを学べた。特に、「土台を作ってオブジェクトを画面に出力すること」が基本になってゲームができていることが理解できた。このことを意識しないと、オブジェクトが画面外に消えてしまったり、不自然な動きをしてしまう。また、オブジェクトの大きさや座標、動く速度などのパラメータ調整も非常に時間がかかり、様々な工夫が必要になることがわかった。さらに、画像や音声をできるだけ軽くしたり、フレームレートを下げたりして、ユーザーが快適にプレイできるような環境作りも注力した。いわゆる「凄い機能」を付けるのではなく、**シンプルでわかりやすく、ユーザーが面白いものを作ること**がゲーム作りの重要なポイントであるとわかった。