```
# ボールの接触判定
# ボール・クラスのリストにあるオブジェクトと指定の円情報との接触判定
# キューとボールの接触判定に使う
# 引数：X, Y, 半径, ボール番号
def BallColliderXYR(x, y, r, num):
    # コリジョン
    wx = balls[num].x - x
    wy = balls[num].y - y
    wl = balls[num].r + r
    if (wx**2 + wy**2) < wl**2:
        return True
    return False


```

```
1 if BallColliderXYR(tipPosX, tipPosY, 10, 0):
2     modeReq = MODEMOVING    # 移動モードに切り替え
3
```

```
    else:   # キューは伸びてない
        tipPosXOld = tipPosX
        tipPosYOld = tipPosY
        rad = math.atan2(clickPosY - mousePosY, clickPosX - mousePosX)
        tipPosX = math.cos(rad) * cueLen + mousePosX
        tipPosY = math.sin(rad) * cueLen + mousePosY
        wx = tipPosX - tipPosXOld
        wy = tipPosY - tipPosYOld
        tipRad = math.atan2(wy, wx)          # チップの向き
        tipSpeed = math.sqrt(wx**2 + wy**2) / deltaTime # チップの速度

```

```
# ボール・クラスのリストにあるオブジェクトへの特定円による物理処理
# 引数：x, y, 半径,向き,速度,ボール番号
def BallForce(x, y, rad, speed, dst):
    wx = balls[dst].x - x
    wy = balls[dst].y - y
    wr = math.atan2(wy,wx)
    ws = math.cos(wr - rad) * speed
    balls[dst].speed = ws
    balls[dst].rad = wr

```

```
if BallColliderXYR(tipPosX, tipPosY, 10, 0):
      BallForce(tipPosX, tipPosY, tipRad, tipSpeed, 0) # ボール加速
      modeReq = MODEMOVING    # 移動モードに切り替え

```

```
FOOTSPOT_X = 600
FOOTSPOT_Y = CENTER_Y
BREAKSHOT_X = 1000
BREAKSHOT_Y = 450

def BallSetup():
    balls.clear()
    balls.append(cBall(BREAKSHOT_X, BREAKSHOT_Y, 0, True))
# 白手玉
    balls.append(cBall(FOOTSPOT_X, FOOTSPOT_Y, 1, True))                                          # 1頂点
    balls.append(cBall(FOOTSPOT_X - FOOTOFFSET_X * 4, FOOTSPOT_Y + FOOTOFFSET_Y *  0, 2, True)) # 2逆頂
点
    balls.append(cBall(FOOTSPOT_X - FOOTOFFSET_X * 2, FOOTSPOT_Y + FOOTOFFSET_Y * -2, 3, True)) # 3右頂
点
    balls.append(cBall(FOOTSPOT_X - FOOTOFFSET_X * 2, FOOTSPOT_Y + FOOTOFFSET_Y *  2, 4, True)) # 4左頂
点
    balls.append(cBall(FOOTSPOT_X - FOOTOFFSET_X * 1, FOOTSPOT_Y + FOOTOFFSET_Y * -1, 5, True)) # 5
    balls.append(cBall(FOOTSPOT_X - FOOTOFFSET_X * 1, FOOTSPOT_Y + FOOTOFFSET_Y *  1, 6, True)) # 6
    balls.append(cBall(FOOTSPOT_X - FOOTOFFSET_X * 3, FOOTSPOT_Y + FOOTOFFSET_Y * -1, 7, True)) # 7
    balls.append(cBall(FOOTSPOT_X - FOOTOFFSET_X * 3, FOOTSPOT_Y + FOOTOFFSET_Y *  1, 8, True)) # 8
    balls.append(cBall(FOOTSPOT_X - FOOTOFFSET_X * 2, FOOTSPOT_Y + FOOTOFFSET_Y *  0, 9, True)) # 9中央

```

```
# ボール・クラスのリストにあるオブジェクト同士の接触判定
def BallColliderList(src, dst):
    wx = balls[dst].x - balls[src].x
    wy = balls[dst].y - balls[src].y
    wl = balls[dst].r + balls[src].r
    if (wx**2 + wy**2) < wl**2:
        return True
    return False

```

```
    # ボールの移動と反射
    cnt = 0 # 処理中のボール番号
    for ball in balls:  # 全てのボールを1個ずつ処理
        if ball.view:   # 表示状態
            screen.blit(sBall[ball.num], (ball.x - ball.r, ball.y - ball.r))     # ボール描画

            # 他のボールとの接触判定
            for num in range(len(balls)):    # 全てのボール番号を取得
                if num != cnt:           # 処理中のボールでない事をチェック
                    if balls[num].view:
                        if BallColliderList(cnt, num):
                            BallForceList(cnt, num) # ボール加速
        cnt += 1    # 次に処理するボール番号に変更

```

```
# ボール・クラスのリストにあるオブジェクト同士の物理処理
def BallForceList(src, dst):
    if balls[src].speed == 0:    # 自身の速度が0なら相手への影響無し
        return
    wx = balls[dst].x - balls[src].x    # 相手ボールとのX座標の差を抽出
    wy = balls[dst].y - balls[src].y    # 相手ボールとのY座標の差を抽出
    wr = math.atan2(wy,wx)              # 座標の差から向きRadianを算出
    ws = math.cos(wr - balls[src].rad) * balls[src].speed    # 接触面の自身の角度成分（速度）のみを抽出
   if ws <= 0: # 速度が0以下は相手に影響を与えない
       return
    ws2 = math.cos(wr - balls[dst].rad) * balls[dst].speed    # 接触面の相手の角度成分（速度）のみを抽出
    if ws2 > ws:    # 接触面の速度が相手より低ければ相手に影響を与えない
        return
    ws3 = ws - ws2  # 相対速度を求める
    if ws3 >= ws:
        ws3 = ws
    balls[dst].ax += math.cos(wr) * ws3  # 加速度成分のXを相手に加算集計
    balls[dst].ay += math.sin(wr) * ws3  #               Y
    balls[src].ax -= math.cos(wr) * ws3  # 相手に与えた加速度の分，自身に逆向きX加速度を加算
    balls[src].ay -= math.sin(wr) * ws3  #                                   Y
```

```
# ブロック（クッション）の管理クラス
class cBlock:
    # コンストラクタ(このクラスが実体化した時の処理)
    def __init__(self, u, d, l, r, view):
        self.u = u  # 上座標(Up)
        self.d = d  # 下座標(Down)
        self.l = l  # 左座標(Left)
        self.r = r  # 右座標(Right)
        self.view = view    # 可視属性

# ブロック（クッション）の初期化
blocks = []
blocks.append(cBlock(0, CUSHION_THICK, POCKET_WIDTH, CENTER_X - SIDEPOCKET_WIDTH / 2, True))
# 上左
blocks.append(cBlock(0, CUSHION_THICK, CENTER_X + SIDEPOCKET_WIDTH / 2, SCREEN_WIDTH - POCKET_WIDTH,
True))                              # 上右
blocks.append(cBlock(SCREEN_HEIGHT - CUSHION_THICK, SCREEN_HEIGHT, POCKET_WIDTH, CENTER_X -
SIDEPOCKET_WIDTH / 2, True))                # 下左
blocks.append(cBlock(SCREEN_HEIGHT - CUSHION_THICK, SCREEN_HEIGHT, CENTER_X + SIDEPOCKET_WIDTH / 2,
SCREEN_WIDTH - POCKET_WIDTH, True)) # 下右
blocks.append(cBlock(POCKET_WIDTH, SCREEN_HEIGHT - POCKET_WIDTH, 0, CUSHION_THICK, True))
# 左
blocks.append(cBlock(POCKET_WIDTH, SCREEN_HEIGHT - POCKET_WIDTH, SCREEN_WIDTH - CUSHION_THICK,
SCREEN_WIDTH, True))                    # 右

# ブロックの表示
for block in blocks:
    if block.view:
        screen.fill((0, 110, 10), (block.l, block.u, block.r - block.l, block.d - block.u))
```

```
# ボール・クラス・リストとブロック・クラス・リストのオブジェクト同士の接触判定と反射処理
# 角のある壁に当たった処理に使う
def BlockRefrectList(ballNum, blockNum):
    # 必要な要素を一時変数にコピー
    ball = balls[ballNum]
    block = blocks[blockNum]
    br = ball.r
    bx = ball.x
    by = ball.y
    u = block.u
    d = block.d
    l = block.l
    r = block.r
    # 明らかに接触範囲外なら反射処理終了
    if bx + br < l: # ブロックより左
        return
    if bx - br > r: # ブロックより右
        return
    if by + br < u: # ブロックより上
        return
    if by - br > d: # ブロックより下
        return
    # ボールの移動成分を取り出す
    speed = ball.speed
    rad = ball.rad
    wx = math.cos(ball.rad) * ball.speed
    wy = math.sin(ball.rad) * ball.speed
    # 左右の辺へ接触か？
    if by >= u and by <= d: # 高さの範囲一致
```

```
        if bx < (l + r) / 2:     # 左の辺
            ball.x = l - br      # 左辺まで押し返す
        else:                    # 右の辺
            ball.x = r + br      # 右辺まで押し返す
        ball.rad = math.atan2(wy, -wx)  # X反転した角度を算出
    # 上下の辺へ接触か？
    if bx >= l and bx <= r: # 左右の範囲一致
        if by < (u + d) / 2:     # 上の辺
            ball.y = u - br      # 上辺まで押し返す
        else:                    # 下の辺
            ball.y = d + br      # 下辺まで押し返す
        ball.rad = math.atan2(-wy, wx)  # X反転した角度を算出
    # 四隅の角へ接触か？
    wrr = br ** 2   # あらかじめボール半径の2乗を計算しておく（三平方の定理用）
    # 左上の接触判定
    if PointRefrect(ball, l, u):
        return
    # 右上の接触判定
    if PointRefrect(ball, r, u):
        return
    # 左下の接触判定
    if PointRefrect(ball, l, d):
        return
    # 右下の接触判定
    if PointRefrect(ball, r, d):
        return

```

```
# 点との接触判定と反射
# 引数：ballオブジェクト, 点X, 点Y
# 戻り値:反射した=True
def PointRefrect(ball, px, py):
    speed = ball.speed
    rad = ball.rad
    x = ball.x
    y = ball.y
    r = ball.r
    wrr = r ** 2   # あらかじめボール半径の2乗を計算しておく（三平方の定理用）
    wdx = px - x    # Xの差分
    wdy = py - y    # Yの差分
    # 接触判定
    if wdx ** 2 + wdy ** 2 < wrr:    # 点と接触したか？
        wrad = math.atan2(wdy,wdx)      # 座標の差から角の法線Radianを算出
        ball.rad = ((wrad - rad) * 2.0 + rad + math.pi)    # ボールの反射
        return True
    return False

```

```
SCREEN_WIDTH = 1600                         # ゲーム画面の幅
SCREEN_HEIGHT = 900                         # ゲーム画面の高さ
POCKET_THICK = 80                           # ポケットの有効厚み

    wpocket = False # ポケット・インの監視
    if ball.x < POCKET_THICK:
        wpocket = True
    elif ball.x > SCREEN_WIDTH - POCKET_THICK:
        wpocket = True
    if ball.y < POCKET_THICK:
        wpocket = True
    elif ball.y > SCREEN_HEIGHT - POCKET_THICK:
        wpocket = True
    # ポケットに入った時の処理
    if wpocket:
        ball.view = False
        if ball.num == 0:
            foulFlag = True # ファール（失敗）
        elif ball.num == 9:
            clearFlag = True    # クリア（勝利）

```

```
    elif mode == MODEMOVING:    # 移動モード
        if foulFlag:
            modeReq = MODEFOUL
        elif movingFlag == False:
            if clearFlag:
                modeReq = MODECLEAR
            else:
                modeReq = MODEIDLE
        else:
            movingFlag = False
    elif mode == MODECLEAR:     # クリア・モード
        messageL = "Win!"
        messageS = "思考時間:{0:.1f}秒".format(thinkTime)
        popupFlag = True
        if mouseButton:
            BallSetup() # ボールの初期化
            modeReq = MODEIDLE
            popupFlag = False
            clearFlag = False
    elif mode == MODEFOUL:      # ファール・モード
        messageL = "Foul!"
        messageS = "思考時間:{0:.1f}秒".format(thinkTime)
        popupFlag = True
        if mouseButton:
            BallSetup() # ボールの初期化
            modeReq = MODEIDLE
            popupFlag = False
            foulFlag = False
```

```
    # ショット数表示
    messageS = "Shot:{0:1}回".format(shotCount)
    rTextS = fontS.render(messageS, True, (220, 240, 100))
    screen.blit(rTextS, (POCKET_WIDTH + 10, 0))

    # ファール数表示
    messageS = "Foul:{0:1}回".format(foulCount)
    rTextS = fontS.render(messageS, True, (220, 240, 100))
    screen.blit(rTextS, (CENTER_X + SIDEPOCKET_WIDTH / 2 + 10, 0))

    # ポケット数表示
    messageS = "Pocket:{0:1}回".format(pocketCount)
    rTextS = fontS.render(messageS, True, (220, 240, 100))
    screen.blit(rTextS, (POCKET_WIDTH + 10, SCREEN_HEIGHT - CUSHION_THICK))

    # 思考時間表示
    messageS = "{0:.1f}秒".format(thinkTime)
    rTextS = fontS.render(messageS, True, (220, 240, 100))
    screen.blit(rTextS, (CENTER_X + SIDEPOCKET_WIDTH / 2 + 10, SCREEN_HEIGHT - CUSHION_THICK))

```

```
thinkTime = 0        # 思考時間
shotCount = 0        # ショット回数
foulCount = 0        # ファール回数

～略～

# ゲームのループ
while True:

～略～

    elif mode == MODERETRY:     # リトライ・モード
        BallSetup() # ボールの初期化
        popupFlag = False   # ポップ・アップの停止
        foulFlag = False    # 手玉ロスト・ファールの初期化
        clearFlag = False   # 勝利の初期化
        thinkTime = 0       # 思考合計時間を０に
        shotCount = 0       # ショット回数
        foulCount = 0       # ファール回数
        modeReq = MODEIDLE  # アイドル・モードに戻る

```

```
    # ボールの移動と反射
    moveCount = 0   # 移動カウント
    stopCount = 0   # 停止カウント
    pocketCount = 0 # ポケット・カウント
    for ball in balls:
        if ball.view:   # 表示状態

～略～

            # ボールの停止・動作数のカウント
            if ball.speed == 0:
                stopCount += 1  # 停止ボールのカウント
            else:
                moveCount += 1  # 移動ボールのカウント

～略～

        else:   # 非表示（ポケット・イン状態）
            pocketCount += 1    # ポケット・インしたボール・カウント

```

```
    elif mode == MODEMOVING:    # 移動モード
        if foulFlag:            # 手玉ロスト・ファール・フラグで強制終了
            modeReq = MODEFOUL  # ファール・モードに切り替え
        elif moveCount == 0:    # 全てのボールの停止か
〜略〜
                modeReq = MODEIDLE  # アイドル・モードに戻る

```

```
    # 動作モードごとの処理分岐
〜略〜
    elif mode == MODEIDLE:  # アイドル・モード

        thinkTime += deltaTime  # 思考時間カウント・アップ

        if mouseButton: # マウス・ボタンが押されたので
            modeReq = MODEBRIDGE    # 次はブリッジ・モード

    elif mode == MODEBRIDGE:    # ブリッジ・モード
        thinkTime += deltaTime  # 思考時間カウント・アップ

```

```
        # コリジョン
        if BallColliderXYR(tipPosX, tipPosY, TIP_RADIUS, 0):     # 手玉接触判定
            BallForce(tipPosX, tipPosY, tipRad, tipSpeed, 0)     # 手玉加速
            shotCount += 1           # ショット回数をカウント・アップ
            inCount = 0              # ポケット・イン回数（ノーポケット・ファール判定用）
            modeReq = MODEMOVING     # 移動モードに切り替え

```

```
            # ポケットに入った時の処理
            if wpocket:
                ball.view = False       # ボールを非表示
                inCount += 1            # 今ショットのポケット・イン数カウント

〜略〜

    elif mode == MODEMOVING:    # 移動モード
        if foulFlag:            # 手玉ロスト・ファール・フラグで強制終了
            modeReq = MODEFOUL  # ファール・モードに切り替え
        elif moveCount == 0:    # 全てのボールの停止か
            if clearFlag:       # クリア条件のフラグがあればクリア・モードに
                modeReq = MODECLEAR
            else:               # 続行状態
                if inCount == 0:    # 1個もポケットに入らなかった
                    foulCount += 1  # ファール数のカウント・アップ

```

```
    # 動作モードごとの処理分岐
〜略〜
    elif mode == MODEIDLE:  # アイドル・モード
        if modeFrame == 0:  # アイドル最初の処理
〜略〜
            # 最小の的玉を探す
            minNum = 0  # 最小番号０（無効）
            for ball in balls:
                if ball.view:
                    if ball.num != 0:   # 手玉は除外
                        if minNum == 0 or ball.num < minNum:    # 最初のボールか小さい的玉番号判定
                            minNum = ball.num   # 最小的玉番号の更新

```

```
            # 他のボールとの接触判定
            for num in range(len(balls)):
                if num != cnt:  # 違うボールだけ判定
                    if balls[num].view: # ポケット・インしてない
                        if BallColliderList(cnt, num):  # ボール接触
                            if minNum != 0: # 手玉が最初に最小的玉に当たった判定が必要か？
                                if balls[num].num != minNum:    # 他の手玉に当てた
                                    foulCount += 1              # ファール数のカウント・アップ
                                minNum = 0  # 手玉の最小的玉判定要求を停止
                            BallForceList(cnt, num) # 当てたボールを加速

```

```
# 効果音の準備
# 音声データのリスト化
se = []
se.append(pygame.mixer.Sound("se/break.mp3"))    # ブレイク・ショット
se.append(pygame.mixer.Sound("se/hit.mp3"))      # ボール同士の反発
se.append(pygame.mixer.Sound("se/pocket.mp3"))   # ポケット・イン
se.append(pygame.mixer.Sound("se/block.mp3"))    # クッション（ブロック）
se.append(pygame.mixer.Sound("se/win.mp3"))      # 勝利
se.append(pygame.mixer.Sound("se/lose.mp3"))     # 敗北
# 音声の要求コード定義
SENON = 0        #
SEBREAK = 1      # ブレイク・ショット
SEHIT = 2        # ボール同士の反発
SEPOCKET = 3     # ポケット・イン
SEBLOCK = 4      # クッション（ブロック）
SEWIN = 5        # 勝利
SELOSE = 6       # 敗北
seReq = SENON    # 音声リクエスト
seHold = 0       # 音声リクエスト禁止時間
seHoldReq = 0    # 音声リクエスト禁止時間要求

～略～

# ゲームのループ
while True:

～略～

    # 効果音の再生
```

```
    if seHold <= 0:                 # 効果音受付可能
        if seReq != SENON:          # 効果音要求の確認
            se[seReq - 1].play()    # 効果音の再生
            seReq = SENON           # 効果音要求のクリア
            if seHoldReq != 0:      # 音声リクエスト禁止時間要求
                seHold = seHoldReq  # 音声リクエスト禁止時間セット
                seHoldReq = 0       # 禁止時間要求のクリア
    else:                           # 効果音要求禁止時間
        seHold -= deltaTime         # 禁止時間のカウント・ダウン

```

```
〜ブレイク・ショット音〜
        # コリジョン
        if BallColliderXYR(tipPosX, tipPosY, TIP_RADIUS, 0):    # 手玉接触判定
            BallForce(tipPosX, tipPosY, tipRad, tipSpeed, 0)    # 手玉加速
            shotCount += 1          # ショット回数をカウントアップ
            inCount = 0             # ポケット・イン回数（ノーポケットファール判定用）
            seReq = SEBREAK         # ブレイク・ショット効果音の要求
            modeReq = MODEMOVING    # 移動モードに切り替え

〜ボール反発音〜
            # 他のボールとの接触判定
            for num in range(len(balls)):
                if num != cnt:  # 違うボールだけ判定
                    if balls[num].view: # ポケット・インしてない
                        if BallColliderList(cnt, num):  # ボール接触
〜略〜
                            seReq = SEHIT           # ボール接触効果音の再生要求

〜ポケット・イン音〜
            # ポケットに入った時の処理
            if wpocket:
                ball.view = False       # ボールを非表示
                inCount += 1            # 今ショットのポケット・イン数カウント
〜略〜
                seReq = SEPOCKET        # ポケット・イン効果音の再生要求

〜クッション（ブロック）音〜
            # ブロックとの接触判定
            for num in range(len(blocks)):
```

```
                if blocks[num].view:
                    if BlockRefrectList(cnt, num):  # 指定ブロックとの接触処理
                        seReq = SEBLOCK             # ブロック反発効果音の要求

〜勝利音〜
    elif mode == MODECLEAR:     # クリア・モード
        if modeFrame == 0:      # 開始時の処理
            seReq = SEWIN       # 勝利音声の要求

〜敗北音〜
    elif mode == MODEFOUL:      # ファール・モード
        if modeFrame == 0:      # 開始時の処理
            seReq = SELOSE      # 敗北音声の要求
            seHoldReq = 0.2     # ボール移動中なので追加音声の禁止時間要求

```